

# REPORT DOCUMENTATION PAGE

Form Approved  
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)

2. REPORT DATE

1998

3. REPORT TYPE AND DATES COVERED

Final, 1991-1998

4. TITLE AND SUBTITLE

Research on Sequential and Parallel Algorithm Synthesis

5. FUNDING NUMBERS

N00014-90-J-1855

6. AUTHOR(S)

M.A. Langston

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(ES)

University of Tennessee  
201 Andy Holt Tower  
Controller's Office  
Knoxville, TN 37996-3104 6100

8. PERFORMING ORGANIZATION  
REPORT NUMBER

113-1A

9. SPONSORING / MONITORING AGENCY NAMES(S) AND ADDRESS(ES)

Office of Naval Research  
100 Alabama St, NW  
Suite 4R15  
Atlanta, GA 30303-3104

10. SPONSORING / MONITORING  
AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

a. DISTRIBUTION / AVAILABILITY STATEMENT

12. DISTRIBUTION CODE

Approved for Public  
Release; Distribution is  
Unlimited

13. ABSTRACT (Maximum 200 words)

The objective of this project has been to perform basic research in the design and analysis of algorithms. The focus has been on fundamental questions in computer software and systems research. Emphasis has been placed largely on developing new approaches to prototypical problems for which only the existence of asymptotically efficient methods was previously known.

Work has proceeded along several fronts, most notably

- 1) well-quasi-order theory and it's application to emergent architectural paradigms,
- 2) fast obstruction tests and their use in novel decision and search algorithms
- 3) parallelization strategies to optimize multiple resources simultaneously and
- 4) techniques for parameter approximation using graph width metrics.

~~Summary of results is given in over 40 refereed journal and conference papers. A bibliography of these papers is attached.~~

14. SUBJECT TERMS

Constructive Complexity, Well-Quasi-Order theory, Parallel Algorithms

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION  
OF REPORT

UNCLASSIFIED

18. SECURITY CLASSIFICATION  
OF THIS PAGE

UNCLASSIFIED

19. SECURITY CLASSIFICATION  
OF ABSTRACT

UNCLASSIFIED

20. LIMITATION OF ABSTRACT

19990402059

## Fast Algorithms for $K_4$ Immersion Testing<sup>\*,†</sup>

Heather D. Booth

*Department of Computer Science, University of Tennessee, Knoxville, Tennessee 37996*

Rajeev Govindan

*Qualcomm Incorporated, San Diego, California 92121*

Michael A. Langston

*Department of Computer Science, University of Tennessee, Knoxville, Tennessee 37996*

and

Siddharthan Ramachandramurthi

*LSI Logic Corporation, Waltham, Massachusetts 02451*

Received February 13, 1996; revised August 15, 1998

Many useful classes of graphs can in principle be recognized with finite batteries of obstruction tests. One of the most fundamental tests is to determine whether an arbitrary input graph contains  $K_4$  in the immersion order. In this paper, we present for the first time a fast, practical algorithm to accomplish this task. We also extend our method so that, should an immersed  $K_4$  be present, a  $K_4$  model is isolated. © 1999 Academic Press

### Contents.

1. Introduction.
2. Preliminaries.
  - 2.1. Three-edge connectivity.
  - 2.2. Series-parallel graph.

\*This research has been supported in part by the National Science Foundation under Grant CDA-9115428 and by the Office of Naval Research under Contract N00014-90-J-1855.

†A preliminary version of a portion of this paper was presented at the Great Lakes Symposium on VLSI, held in Kalamazoo, MI in February, 1992.



3. *Testing for  $K_4$ .*
  - 3.1. *Algorithm decompose.*
  - 3.2. *Algorithm components.*
  - 3.3. *The correctness of components.*
  - 3.4. *Algorithm test.*
  - 3.5. *The correctness of test.*
4. *Finding a model.*
  - 4.1. *Algorithm corners.*
  - 4.2. *Algorithm paths.*
5. *Discussion.*
- References.*

## 1. INTRODUCTION

We restrict our attention to finite, undirected graphs. Multiple edges may be present, but loops are ignored. A pair of adjacent edges  $uv$  and  $vw$ , with  $u \neq v \neq w$ , is *lifted* by deleting the edges  $uv$  and  $vw$ , and by adding the edge  $uw$ . A graph  $H$  is *immersed* in a graph  $G$  if and only if a graph isomorphic to  $H$  can be obtained from  $G$  by taking a subgraph and by lifting pairs of edges.

The immersion order can be applied to a number of combinatorial problems. Consider, for example, the problem of deciding whether a graph satisfies a given width metric. The *cutwidth* of  $G = (V, E)$  is the minimum, over all linear layouts of  $V$ , of the maximum, over all pairs  $u$  and  $v$  of consecutive vertices, of the number of edges from  $E$  that must be cut to split the layout between  $u$  and  $v$ . Although  $\mathcal{NP}$ -complete in general, cutwidth can, in principle, be decided in linear time for any fixed width using a finite but unknown list of immersion tests. Multidimensional generalizations of cutwidth, termed *congestion* problems, can likewise be solved in linear time if only one has the right collection of immersion tests available. These and other problems amenable to the immersion order arise during circuit fabrication, parallel computation, network design and many other processes.

The graphs required for the aforementioned tests are called *obstructions*. So, for example, when one knows all obstructions to cutwidth  $k$ , one knows a characterization for the family of graphs that have cutwidth  $k$  or less. Given the right collection of obstructions, linear-time decidability is assured by bounding an input graph's treewidth [9], computing its tree decomposition [2], and applying dynamic programming to test each obstruction against the decomposition [19]. We refer the reader to [8] for detailed information on this subject.

Unfortunately, little is known about immersion obstructions in general or about practical immersion tests in particular. Complete graphs are often obstructions. Testing for  $K_1$  and  $K_2$  are trivial. Detecting a  $K_3$  is easy:  $K_3$  is immersed in any graph of order 3 or more unless the graph is a tree with no pair of multiple edges incident on a common vertex.

The first really difficult test, and the one we devise here, is for  $K_4$ . Observe that  $K_4$  is an obstruction for cutwidth 3, because any arrangement of its vertices on a line requires a cut of four edges. Ours is the first practical linear-time algorithm known for this task.

If a graph contains a topological  $K_4$ , then it also contains an immersed  $K_4$ . Thus we consider only those graphs with no topological  $K_4$ . These are exactly the series-parallel graphs [4]. But  $K_4$  can be immersed in a series-parallel graph. As a simple example, consider the star graph with three rays, each ray with three edges, as shown in Fig. 1. Clearly, multiple edges are critical, making immersion tests potentially more complicated than tests in the more familiar minor and topological orders (see, for example, [15]).

In the next section we state relevant definitions and we derive a few useful technical lemmas. In Sections 3 and 4, we present algorithms for  $K_4$  immersion testing and  $K_4$  model finding, respectively. Although many of the explanatory details are tedious, especially the correctness proofs, the algorithms themselves are straightforward to implement. In a final section we discuss efficiency, applications and parallelization.

## 2. PRELIMINARIES

We concentrate on edge-disjoint paths, which are relevant due to the following alternate characterization of immersion containment:  $H = (V_H, E_H)$  is immersed in  $G = (V_G, E_G)$  if and only if there exists an injection from  $V_H$  to  $V_G$  for which the images of adjacent elements of  $V_H$  are connected in  $G$  by edge-disjoint paths. Under such an injection, an image vertex is called a *corner* of  $H$  in  $G$ ; all image vertices and their associated paths are collectively called a *model* of  $H$  in  $G$ . Our algorithms exploit the edge connectivity of the input graph.

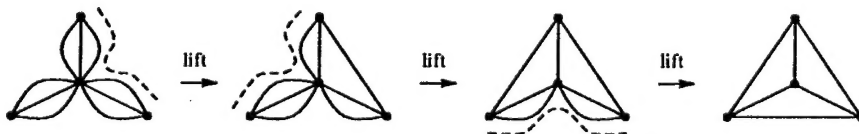


FIG. 1. A series-parallel graph with an immersed  $K_4$ .

### 2.1. Three-Edge Connectivity

A *cut point* in a connected graph  $G$  is a vertex whose removal disconnects  $G$ . Two vertices in  $G$  are said to be *biconnected* if they cannot be disconnected by the removal of any cut point. A *biconnected component* of  $G$  is the subgraph induced by a maximal set of pairwise biconnected vertices.

A *cut edge* in  $G$  is an edge whose removal disconnects  $G$ . A pair of edges, neither of which is a cut edge, is said to form a *cut edge pair* if removing both of them disconnects  $G$ . Two vertices are *three-edge-connected* if there are at least three edge-disjoint paths between them.  $G$  is *three-edge-connected* if and only if it has no cut edges and no cut edge pairs.

A *three-edge-connected component* of  $G = (V, E)$  is a graph  $G' = (V', E')$  where  $V' \subseteq V$  is a maximal set of vertices that are pairwise three-edge-connected in  $G$ .  $E'$  contains all edges induced by  $V'$  plus a (possibly empty) set of *virtual edges* defined as follows: for  $\{u, v\} \subseteq V'$ , a virtual edge  $uv$  is added to  $E'$  for each distinct  $\{x, y\} \subseteq V - V'$  such that  $ux$  and  $vy$  form a cut edge pair in  $G$ . Note that, due to the possible preference of virtual edges, a three-edge-connected component will not necessarily be a subgraph.

LEMMA 1. *If  $K_4$  is immersed in  $G$ , then  $K_4$  is immersed in some three-edge-connected component of  $G$ .*

*Proof.* Let  $a, b, c$ , and  $d$  denote the corners of a  $K_4$  model in  $G$ . These corners are joined (in  $G$ ) by at least six edge-disjoint paths:  $[ab]$ ,  $[ac]$ ,  $[ad]$ ,  $[bc]$ ,  $[bd]$ , and  $[cd]$ . Thus  $a$  and  $b$  are connected by at least three edge-disjoint paths:  $[ab]$ ,  $[ac][cb]$ , and  $[ad][db]$ . Maximality ensures that the three-edge-connected component containing  $a$  also contains  $b$  and, by symmetry,  $c$  and  $d$ . Let  $G_a$  denote this component. If  $[ab]$  contains edges not in  $G_a$ , then  $[ab]$  can be written as  $[au]ux[xy]yv[vb]$ , where  $ux$  and  $yv$  are a cut edge pair in  $G$  and  $uv$  is a virtual edge in  $G_a$ . Thus  $a$  and  $b$  are connected within  $G_a$  by  $[au]uv[vb]$ , which is edge disjoint from the other five paths of the model. By symmetry, all pairs of corners are so connected within  $G_a$ . ■

The proof of Lemma 1 can be generalized to any three-edge-connected graph immersed in another.

For our purposes, a multigraph is said to be *reduced* if all but four copies of any edge having multiplicity 5 or more are removed.

LEMMA 2. *If  $K_4$  is immersed in  $G$ , then  $K_4$  is immersed in the reduced graph of  $G$ .*

*Proof.* Let  $a, b, c$ , and  $d$  denote the corners of a  $K_4$  model in  $G$ , and suppose five or more copies of the edge  $uv$  are contained within its six edge-disjoint paths. Without loss of generality, assume these paths are simple. For some pair of corners, say  $a$  and  $b$ , all five paths with an endpoint at  $a$  or  $b$  contain both  $u$  and  $v$ . Either  $u$  is a corner, or it can be made a corner by replacing  $a$  with  $u$  (deleting the three subpaths of the form  $[au]$ ). Similarly, either  $v$  is a corner or  $b$  can be replaced with  $v$ .  $G$  therefore contains a  $K_4$  model with corners  $u, v, w$ , and  $x$ , where  $\{w, x\} \subset \{a, b, c, d\}$ . At most one of  $[uw], [vw]$  must contain  $uv$ ; at most one of  $[ux], [vx]$  must contain  $uv$ . Thus, of the six edge-disjoint paths of this model, at least two need not contain  $uv$ , and all but four copies of  $uv$  can be eliminated. This construction is iterated until a model is obtained whose edges each have at most four copies. Edges not in this model are now removed until  $G$  is reduced. ■

In the sequel, we assume that all graphs are reduced.

## 2.2. Series-Parallel Graphs

Series-parallel graphs have been widely studied, and are characterizable in several ways. As mentioned in Section 1, one such characterization relies on the absence of a topological  $K_4$ . Topological containment can be defined as a restricted form of immersion containment, with lifting permitted only at vertices of degree 2. Alternately, topological containment can be viewed as an injection, but with vertex-disjoint rather than edge-disjoint paths.

**LEMMA 3.** *Each three-edge-connected component of a series-parallel graph is series-parallel.*

*Proof.* The proof is straightforward, by noting that virtual edges introduce no additional vertex-disjoint paths. ■

Another useful characterization is much older, and based on graphs that are said to be *two-terminal series-parallel* (henceforth 2TSP). A 2TSP graph is defined in terms of base graphs and two types of composition operators. A base graph is a copy of  $K_2$ , with vertices (terminals) labeled "source" and "sink." A series operator combines two graphs by identifying one's source with the other's sink. A parallel operator combines two graphs by identifying source with source and sink with sink. Hence the characterization: a graph is series-parallel if and only if its biconnected components are two-terminal series-parallel.

This characterization is often attractive because it prompts a natural "decomposition tree"  $T$  whose labels indicate how a 2TSP graph can be broken back down into base graphs and operators. If a 2TSP graph is

merely a base graph  $e$ ,  $T$  is a single vertex with label  $e$ . Otherwise,  $T$  is formed from the decomposition trees,  $T_1$  and  $T_2$ , of the pair of 2TSP graphs used in the composition. The roots of  $T_1$  and  $T_2$  are joined to the root of  $T$ , which is labeled  $S$  in the case of a series composition and  $P$  in the case of a parallel composition. Nodes labeled  $S$  and  $P$  are termed  $S$ -nodes and  $P$ -nodes, respectively.

We conclude this section by noting from [3] that if a simple graph  $H$  is series-parallel, then  $|E_H| \leq 2|V_H| - 3$ . From this bound and Lemma 2, we know that all graphs of interest have at most a linear number of edges.

### 3. TESTING FOR $K_4$

Let  $G$  denote an arbitrary input graph with  $n$  vertices and  $m$  distinct edges. Without loss of generality, we assume  $G$  has already been reduced and is input as a simple graph with integer weights indicating edge multiplicities.

Our method to test for the presence of an immersed  $K_4$  proceeds in three steps. Algorithm *decompose* is first invoked to determine whether  $G$  is series-parallel. If  $G$  is series-parallel, then algorithm *components* is used to break  $G$  into three-edge-connected components. Finally, algorithm *test* is employed to search each three-edge-connected component separately for an immersed  $K_4$ .

#### 3.1. Algorithm *decompose*

Algorithm *decompose* is modeled on the method of [11]. It determines whether  $G$  is series-parallel and, if so, computes a decomposition tree for each biconnected component. To accomplish this, *decompose* makes use of the fact that for any edge  $st$  in a biconnected graph  $B$  with  $p$  vertices, the vertices of  $B$  may be numbered from 1 to  $p$  so that vertex  $s$  receives number 1, vertex  $t$  receives number  $p$ , and every vertex except  $s$  and  $t$  is adjacent to both a higher numbered vertex and a lower numbered vertex [14]. Such a numbering is called an  $s, t$ -numbering for  $B$ .

**algorithm** *decompose*( $G$ )

input: a multigraph  $G$

output: a series parallel decomposition tree for each biconnected component of  $G$  if  $G$  is series-parallel, NO otherwise

**begin**

  find the biconnected components  $B_1, \dots, B_k$  of  $G$

**for**  $i = 1$  **to**  $k$  **do**

**begin**

      choose a pair of adjacent vertices to be the source  $s$  and sink  $t$  in  $B_i$

```

    find an  $s, t$ -numbering of  $B_i$ 
     $\bar{B}_i :=$  the directed graph obtained by orienting each edge in  $B_i$  from
        the endpoint with the lower  $s, t$ -number to the one with the
        higher number
    if  $\bar{B}_i$  is a directed 2TSP graph
        then compute a series-parallel decomposition tree  $T_i$  for  $B_i$ 
        else output NO and halt
    end
output  $T_1, \dots, T_k$ 
end

```

The correctness of *decompose* is based on the observation that any pair of adjacent vertices may be chosen as  $s$  and  $t$  [11]. Efficient methods for finding biconnected components and computing  $s, t$ -numberings are known [20, 5]. Techniques for determining whether directed graphs are 2TSP and finding decomposition trees can be found in [21]. All these algorithms are linear in  $n$  and  $m$ ; thus *decompose* runs in  $O(n)$  time.

### 3.2. Algorithm components

*Algorithm components* finds the three-edge-connected components of a series-parallel multigraph in linear time. The input to *components* is a series-parallel graph and a series-parallel decomposition tree for each of its biconnected components. The output is its set of three-edge-connected components (including virtual edges).

We proceed by first removing all cut edges. These are easily found because each cut edge is contained in a biconnected component consisting only of that edge. Notice that each cut edge pair must be contained within some biconnected component. Thus it suffices to give an algorithm for computing the three-edge-connected components of a biconnected 2TSP graph.

Let  $G$  be such a 2TSP graph with source  $s$  and sink  $t$ . Let  $e, f$  be a cut edge pair in  $G$ . Let  $G_1$  and  $G_2$  be the graphs left when  $e$  and  $f$  are deleted from  $G$ . We call this cut edge pair  $s, t$ -nonseparating if  $s$  and  $t$  are both in  $G_1$  or both in  $G_2$ . Otherwise we call the pair  $s, t$ -separating. We say an  $s, t$ -nonseparating pair is *special* if its deletion, followed by the addition of virtual edges, results in two graphs such that one contains  $s$  and  $t$  and the other is three-edge-connected.

These definitions are illustrated in Fig. 2. In this figure, edges  $ab$  and  $cd$  are a special pair of graph  $G$ . Deleting them and adding virtual edges  $ad$  and  $bc$  gives  $G_1$ , which contains both  $s$  and  $t$ , and  $G_2$ , which is three-edge-connected. Edge  $st$  and the virtual edge  $ad$  together form an  $s, t$ -separating pair in  $G_1$ .  $G_{11}$ ,  $G_{12}$ , and  $G_2$  are the three-edge-connected components of  $G$ .

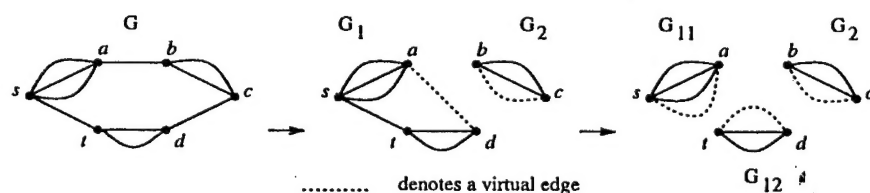


FIG. 2. A two-terminal series-parallel graph with cut edge pairs.

For our purposes, the decomposition tree  $T$  for a 2TSP graph  $G$  must be *ordered*. That is, if  $x$  is a tree node representing a graph formed by composing  $G_1$  and  $G_2$  in series such that the sink of  $G_1$  is identified with the source of  $G_2$ , then the left child of  $x$  must be the root of a decomposition tree for  $G_1$  and the right child of  $x$  must be the root of a decomposition tree for  $G_2$ . Thus the order among children of a series node is fixed. The children of a parallel node can be in any order. Additionally, we assume that an edge  $uv$  stored at a leaf of a decomposition tree is represented by the ordered pair  $(u, v)$ , where  $u$  has a smaller number than  $v$  in the  $s, t$ -numbering used in decompose.

Our algorithm proceeds in two phases. In the first phase special pairs are found and deleted (and appropriate virtual edges are added) until no more are left. This leaves a collection of (isolated vertices and) 2TSP graphs, one of which contains both  $s$  and  $t$ . We will call this graph  $G_{s,t}$ . All other graphs in the collection are three-edge-connected. We can show that  $G_{s,t}$  contains at most one cut edge pair. In the second phase the last remaining cut edge pair, if it exists, is found and removed, and virtual edges are added.

In order to find any of these cut edge pairs we use the *compressed* decomposition tree for the graph. A compressed decomposition tree is formed from a binary decomposition tree merely by identifying all pairs of adjacent nodes that are of the same type.

Let  $G$  be a biconnected 2TSP graph with a compressed decomposition tree  $T$ . Special pairs can be found by processing  $T$  in a bottom-up fashion. When a special pair  $e, f$  is removed, virtual edges are added and  $T$  is modified to represent  $G'_{s,t}$ , the graph containing  $s$  and  $t$  that is left after removing  $e$  and  $f$  from  $G$  (the other graph left is a three-edge-connected component).

Pseudo-code for components is presented in the following text. In a compressed tree, each internal node will have at least two children, stored in a linked list called *child list*. Stored along with each tree node is its type ( $P$ ,  $S$ , or leaf), a pointer to its child list and, if it is a leaf node, an ordered pair giving the endpoints of its associated edge.

The following functions are also used:

**left\_child( $x$ ).** for  $x$  a tree node, if  $x$  is not a leaf, this returns the leftmost child in  $x$ 's child list; otherwise, it returns the value NULL.

**right\_child( $x$ ).** for  $x$  a tree node, if  $x$  is not a leaf, this returns the rightmost child in  $x$ 's child list; otherwise it returns the value NULL.

**next\_sibling( $q$ ).** for  $q$  a nonroot tree node, this returns the child following  $q$  in the child list of the parent of  $q$  or null if no such child exists.

**leftmost\_leaf( $x$ ).** for  $x$  a tree node, if  $x$  is not a leaf, this returns the leftmost node in  $x$ 's child list that is a leaf or NULL if no such node exists.

**algorithm components ( $T$ )**

input: a binary series-parallel decomposition tree  $T$  of a biconnected multigraph  $G$

output: the three-edge-connected components of  $G$

**begin**

$r :=$  the root of  $T$

compress( $r$ )

remove\_non\_sep( $r$ )

remove\_sep( $r$ )

**end**

**algorithm compress( $x$ )**

input: a node  $x$  in a binary series-parallel decomposition tree  $T$

output: the compressed form of the subtree rooted at  $x$

**begin**

**if**  $x$  is a leaf node **then return**

compress(left\_child( $x$ ))

compress(right\_child( $x$ ))

**if**  $x$  and left\_child( $x$ ) are of the same type

**then** in the child list of  $x$ , replace left\_child( $x$ ) by the child list of left\_child( $x$ )

**if**  $x$  and right\_child( $x$ ) are of the same type

**then** in the child list of  $x$ , replace right\_child( $x$ ) by the child list of right\_child( $x$ )

**end**

**algorithm remove\_non\_sep( $q$ )**

input: a node  $q$  in a compressed series-parallel decomposition tree  $T$  of a multigraph  $G$

output: the graph  $G$ , after deletion of all  $s, t$ -nonseparating pairs that are contained in the subtree of  $T$  rooted at  $q$ , and addition of virtual edges

```

begin
  ch := left_child(q)
  while ch is not NULL
    begin
      if ch is not a leaf node then remove_non_sep(ch)
      ch := next_sibling(ch)
    end
  if q is an S-node
    then while q has two children that are leaves
      begin
        leaf1, leaf2 := the first two leaf-node children of q
        (u, v) := the ordered edge associated with leaf1
        (w, x) := the ordered edge associated with leaf2
        delete edges uv and wx from G
        add edges ux and vw to G
        create tree node new representing the ordered edge (u, x)
        if q has more than two children
          then replace all children of q between leaf1 and leaf2 (inclusive) by new
          else replace q by new
        end
      end
    end
end

algorithm remove_sep(root)
input: the root of a compressed series-parallel decomposition tree T of a
       multigraph G without any s, t-nonseparating pairs
output: the graph G after deletion of the s, t-separating pair, if present,
        and addition of virtual edges
begin
  if root has exactly two children
    then begin
      c1, c2 := the children of root
      if c1 is not a leaf node then c1 := leftmost_leaf(c1)
      if c2 is not a leaf node then c2 := leftmost_leaf(c2)
      if c1 and c2 are both nonNULL
        then begin
          (u, v) := the ordered edge associated with c1
          (w, x) := the ordered edge associated with c2
          delete edges uv and wx from G
          add edges uv and ux to G
        end
      end
    end
end

```

LEMMA 4. *Algorithm components runs in  $O(m + n)$  time on a graph with  $n$  vertices and  $m$  edges.*

*Proof.* The algorithm takes time proportional to the size of the binary decomposition tree, which is  $O(m + n)$ . ■

Thus, in our setting, components takes  $O(n)$  time. We note for completeness that a more complex linear-time approach may be viable [18], by modifying the ear decomposition techniques used to decide vertex connectivity in [10].

### 3.3. The Correctness of components

Neither the components driver nor algorithm compress require discussion.

Consider algorithm `remove_non_sep`. Note first that `remove_non_sep` cannot inadvertently remove an  $s, t$ -separating pair, because the edge  $st$  must be a child of the root (which is a  $P$ -node because  $G$  is biconnected), and `remove_non_sep` eliminates only edges that are children of  $S$ -nodes.

To proceed, we classify edges and pairs of edges in a 2TSP graph as follows. A single edge can be either a cut edge or a noncut edge. A pair of edges can be: a pair of cut edges, an  $s, t$ -separating pair, an  $s, t$ -nonseparating pair, or a *noncut pair*.

Let  $G_1$  and  $G_2$  be 2TSP graphs such that  $G_s$  is the graph formed by composing them in series and  $G_p$  is the graph formed by composing them in parallel. Suppose  $e$  is an edge in  $G_1$  and  $f$  is an edge in  $G_2$ . Table 1 shows the relation between the class of edge  $e$  in  $G_1$ , edge  $f$  in  $G_2$ , and the pair  $e, f$  in  $G_s$  and  $G_p$ . For example, if edges  $e$  and  $f$  are cut edges in  $G_1$  and  $G_2$ , respectively, then  $e$  and  $f$  must be an  $s, t$ -separating pair in  $G_p$ .

Now suppose edges  $e$  and  $f$  are both in the 2TSP graph  $G_1$ , and  $G_2$  is any other 2TSP graph. Graphs  $G_s$  and  $G_p$  are as defined previously. Table 2 relates the class of  $e$  and  $f$  in  $G_1$  to their class in  $G_s$  and  $G_p$ .

TABLE 1

Class of edge $e$ in $G_1$	Class of edge $f$ in $G_2$	Class of $e$ and $f$	
		In $G_s$	In $G_p$
noncut edge	noncut edge	noncut pair	noncut pair
noncut edge	cut edge	$f$ a cut edge	noncut pair
cut edge	noncut edge	$e$ a cut edge	noncut pair
cut edge	cut edge	cut edges	$s, t$ -separating

TABLE 2

Class of edges $e$ and $f$		
In $G_1$	In $G_s$	In $G_p$
cut edges	cut edges	$s, t$ -nonseparating
$s, t$ -nonseparating	$s, t$ -nonseparating	$s, t$ -nonseparating
$s, t$ -separating	$s, t$ -separating	noncut pair
noncut pair	noncut pair	noncut pair

Let  $G$  be a 2TSP graph with compressed decomposition tree  $T$ . Let  $x$  denote an arbitrary internal node in  $T$ , and let  $T_x$  denote the subtree of  $T$  rooted at  $x$ . Then the 2TSP graph that has  $T_x$  as a decomposition tree is a *constituent graph* for  $G$  with respect to  $T$ . For any edge  $e$  in  $G$ , let  $\hat{e}$  denote the node in  $T$  with label  $e$ . If  $e$  and  $f$  are edges in  $G$  then the *least constituent graph containing  $e$  and  $f$*  is the smallest constituent graph  $H$  of  $G$  that contains both  $e$  and  $f$ . This graph has a decomposition tree  $T_z$ , where  $z$  is the least common ancestor of  $\hat{e}$  and  $\hat{f}$  in  $T$ . Table 3 gives the relation between the class of an edge in  $H$  and its class in  $G$ .

The following lemmas are used to justify the correctness of the procedure for finding special pairs, removing special pairs, updating the decomposition tree for the connected component containing  $s$  and  $t$ , and adding virtual edges.

**LEMMA 5.** *Let  $G$  be a 2TSP graph, and let  $e$  and  $f$  be an  $s, t$ -nonseparating pair in  $G$ . If  $H$  is the least constituent graph of  $G$  containing  $e$  and  $f$ , then  $e$  and  $f$  are cut edges in  $H$ .*

*Proof.* By the first two lines of Table 3, either  $e$  and  $f$  are cut edges in  $H$ , as claimed, or they form an  $s, t$ -nonseparating pair. Because  $H$  is a least constituent graph,  $H$  must be formed by composing two 2TSP graphs  $H_1$  and  $H_2$  such that  $H_1$  contains  $e$  and  $H_2$  contains  $f$ . According to Table 1,  $e$  and  $f$  cannot be an  $s, t$ -nonseparating pair in  $H$ . Therefore they must be cut edges in  $H$ , as claimed. ■

TABLE 3

Class of edges $e$ and $f$	
In $H$	In $G$
cut edges	cut edges or $s, t$ -nonseparating
$s, t$ -nonseparating	$s, t$ -nonseparating
$s, t$ -separating	$s, t$ -separating or noncut pair
noncut pair	noncut pair

LEMMA 6. If  $G$  is a 2TSP graph and  $T$  is a compressed decomposition tree for  $G$ , then edge  $e$  is a cut edge in  $G$  if and only if the root of  $T$  is an  $S$ -node and  $\hat{e}$  is a child of the root.

*Proof.* Suppose  $e$  is a cut edge in  $G$ . Then no ancestor of  $e$  in  $T$  is a  $P$ -node, because subtrees rooted at  $P$ -nodes represent biconnected graphs. Hence the path from the root of  $T$  to  $e$  includes only  $S$ -nodes. But because  $T$  is compressed,  $e$  must be a child of the root, which must be an  $S$ -node. Conversely, if the root of  $T$  is an  $S$ -node and  $\hat{e}$  is a child of the root, then every path from  $s$  to  $t$  in  $G$  must include  $e$ . Therefore  $e$  is a cut edge. ■

LEMMA 7. If  $e$  and  $f$  are edges in a biconnected 2TSP graph  $G$  with a compressed decomposition tree  $T$ , then  $e$  and  $f$  are an  $s, t$ -nonseparating pair if and only if  $\hat{e}$  and  $\hat{f}$  are siblings whose parent is an  $S$ -node.

*Proof.* Let  $z$  be the least common ancestor of  $\hat{e}$  and  $\hat{f}$  in  $T$ . Let  $H$  be the 2TSP graph having  $T_z$  as a decomposition tree. Note that  $H$  is the least constituent graph of  $G$  that contains  $e$  and  $f$ .

Suppose  $e$  and  $f$  are  $s, t$ -nonseparating. By Lemma 5,  $e$  and  $f$  are cut edges in  $H$ . Then, by Lemma 6,  $\hat{e}$  and  $\hat{f}$  are children of  $z$  and  $z$  is an  $S$ -node.

Now suppose  $\hat{e}$  and  $\hat{f}$  are siblings whose parent is an  $S$ -node. This implies that  $e$  and  $f$  are cut edges in  $H$ . Then, by Table 3,  $e$  and  $f$  must be either cut edges or an  $s, t$ -nonseparating pair in  $G$ . Because  $G$  is biconnected,  $e$  and  $f$  must in fact be an  $s, t$ -nonseparating pair. ■

In what follows, we say that node  $x$  in tree  $T$  occurs "between" nodes  $y$  and  $z$  if  $x$  occurs between  $y$  and  $z$  in the pre-order traversal of  $T$ . Let  $H_x$  denote the graph having  $T_x$  as a decomposition tree.

LEMMA 8. Let  $G$  be a biconnected 2TSP graph and let  $T$  be a compressed decomposition tree for  $G$ . In  $G$ , let  $e$  and  $f$  be an  $s, t$ -nonseparating pair whose removal yields a graph  $G_1$  containing  $s$  and  $t$ , and another graph  $G_2$ . Suppose  $\hat{e}$  occurs before  $\hat{f}$  in  $T$ , and let  $(u, v)$  and  $(w, x)$  be the pairs stored with  $\hat{e}$  and  $\hat{f}$ , respectively. Then the edges in  $G_2$  are  $\{g \mid \hat{g} \text{ occurs between } \hat{e} \text{ and } \hat{f} \text{ in } T\}$ , and the vertices in  $G_2$  are the endpoints of these edges plus  $\{v, w\}$ .

*Proof.* Because  $e$  and  $f$  are  $s, t$ -nonseparating, by Lemma 7,  $\hat{e}$  and  $\hat{f}$  are siblings whose parent  $z$  is an  $S$ -node. Because  $G$  is biconnected,  $z$ 's parent,  $y$ , is a  $P$ -node.

Removal of  $e$  and  $f$  from  $H_z$  leaves three graphs  $H_1$ ,  $H_2$ , and  $H_3$  such that:  $H_1$  contains all edges represented by nodes occurring before  $\hat{e}$  in  $T_z$ , their associated vertices, and vertex  $u$ ;  $H_2$  contains all edges represented by nodes occurring between  $\hat{e}$  and  $\hat{f}$  and associated vertices plus  $\{v, w\}$ ;

and  $H_3$  contains all edges represented by nodes occurring after  $\hat{f}$  in  $T_z$  and associated vertices plus  $x$ . The source and sink of  $H_2$  are in  $H_1$  and  $H_3$ , respectively.

In  $H_y$ ,  $e$  and  $f$  are an  $s, t$ -nonseparating pair whose removal leaves  $H_2$  and another graph containing  $H_1$ ,  $H_3$ , and the portion of  $H_y$  not in  $H_2$ . The source and sink of  $H_y$  are the source and sink of  $H_2$  and are not in  $H_2$ . Thus the claim holds for  $H_y$ .

Any graph formed by composing two 2TSP graphs, one of which has  $H_y$  as a constituent, still has the claimed property because the paths in the new graph that are not in  $H_y$  can only connect vertices not in  $H_2$ . Because no new paths are added from vertices in  $H_2$  to vertices not in  $H_2$ , it is still the case that removal of  $e$  and  $f$  separates the vertices in  $H_2$  from the rest of the graph. Thus the claim also holds for any graph having  $H_y$  as a constituent. ■

**COROLLARY 1.** *Let  $G$ ,  $T$ ,  $e$ , and  $f$  be as defined in Lemma 8. Let  $G'_1$  be the graph consisting of  $G_1$  plus virtual edge  $ux$  and let  $G'_2$  be the graph consisting of  $G_2$  plus virtual edge  $vw$ . Let  $z$  be the parent of  $\hat{e}$  and  $\hat{f}$  in  $T$  and let  $r_1, \dots, r_k$  be the children of  $z$  in order from left to right such that  $r_i = \hat{e}$  and  $r_j = \hat{f}$ . Let  $\hat{g}$  be a tree node representing  $g = ux$ ; the ordered pair stored with  $\hat{g}$  is  $(u, x)$ . Let  $\hat{h}$  be a tree node representing  $h = vw$ ; the ordered pair stored with  $\hat{h}$  is  $(v, w)$ .*

*A decomposition tree for  $G'_1$  is formed by replacing  $r_i, \dots, r_j$  by node  $\hat{g}$  if  $i \neq 1$  or  $j \neq k$  and replacing  $T_z$  by  $\hat{g}$  otherwise.*

*A decomposition tree for  $G'_2$  is one of the following:*

- (a) *empty, if  $j = i + 1$ ;*
- (b) *a P-node with children  $\hat{h}$  and  $r_{i+1}$ , if  $j = i + 2$ ;*
- (c) *a P-node with two children  $\hat{h}$  and an S-node, which in turn has children  $r_{i+1}, \dots, r_{j-1}$ , otherwise.*

*Proof.* We know by Lemma 8 that  $G'_1$  is formed by replacing the portion of  $G$  represented by nodes in  $T$  between  $\hat{e}$  and  $\hat{f}$  by a single edge  $ux$ , so the decomposition tree for  $G'$  is as claimed. We also know that  $G'_2$  consists of the edges represented by nodes in  $T$  strictly between  $\hat{e}$  and  $\hat{f}$ , their associated vertices and vertices  $v$  and  $w$ , with the edge  $vw$  composed in parallel. Because the nodes between  $\hat{e}$  and  $\hat{f}$  are children of an S-node, the decomposition tree for  $G'_2$  is as claimed. ■

**LEMMA 9.** *Let  $G$  be a biconnected 2TSP graph and let  $T$  be a compressed decomposition tree for  $G$ . A pair of  $s, t$ -nonseparating edges,  $e, f$ , is a special pair in  $G$  if and only if for every sibling  $y$  of  $\hat{e}$  and  $\hat{f}$  in  $T$  that occurs between  $\hat{e}$  and  $\hat{f}$ ,  $y$  is not a leaf and  $T_y$  does not represent a graph containing an  $s, t$ -nonseparating pair.*

*Proof.* Because  $e$  and  $f$  are  $s, t$ -nonseparating, removal of  $e$  and  $f$  yields two graphs  $G_1$  and  $G_2$  such that  $G_1$  contains  $s$  and  $t$ . Let  $G'$  be the graph  $G_2$  plus the virtual edge. Edges  $e$  and  $f$  are special if and only if  $G'$  is three-edge-connected. Let  $T'$  be the decomposition tree for  $G'$  as described in Corollary 1. Let  $z$  be the parent of  $\hat{e}$  and  $\hat{f}$  in  $T$ .

Suppose  $e$  and  $f$  are special. We employ proof by contradiction and we assume there exists a child  $y$  of  $z$  between  $\hat{e}$  and  $\hat{f}$  such that  $y$  is a leaf or  $T_y$  represents a graph containing an  $s, t$ -nonseparating pair.  $G'$  must be three-edge-connected, which implies  $T'$  has no cut edges or cut edge pairs. If  $y$  is a leaf then, by Corollary 1,  $T'$  consists of a  $P$ -node with two children. One of them is a leaf (representing the virtual edge) and the other is either  $\hat{y}$  or an  $S$ -node having  $\hat{y}$  as a child. In either case the structure of  $T'$  requires that the virtual edge and  $y$  form an  $s, t$ -separating pair in  $G'$ , a contradiction. If, on the other hand,  $y$  is a nonleaf node whose subtree  $T_y$  represents a graph having an  $s, t$ -nonseparating pair, then  $T_y$  contains an  $S$ -node with two leaves as children. These nodes also represent an  $s, t$ -nonseparating pair in  $G'$ , again a contradiction.

Now suppose  $\hat{e}$  and  $\hat{f}$  satisfy the conditions of the lemma, but that  $e$  and  $f$  are not special. According to Lemma 7,  $z$  is an  $S$ -node. Because  $T$  is compressed and no  $y$  is a leaf, each  $y$  is a  $P$ -node. The root of  $T'$  is a  $P$ -node, implying that  $G'$  has no cut edge. Moreover, the root has exactly two children, one of which is a virtual edge. Let  $x$  denote the other child. If there is only one node  $y$  between  $e$  and  $f$  in  $T$ , then  $x$  is a  $P$ -node with  $T_x = T_y$ ; else  $x$  is an  $S$ -node with all the  $y$  as children. Because each  $T_y$  is rooted at a  $P$ -node, no  $T_y$  has a cut edge and neither does  $T_x$ . Therefore  $G'$  has no  $s, t$ -separating pairs. If  $T'$  has an  $s, t$ -nonseparating pair, then  $T_x$  must have either an  $s, t$ -nonseparating pair or a pair of cut edges. This in turn means that some  $T_y$  contains a cut edge or an  $s, t$ -nonseparating cut edge pair, a contradiction. ■

**COROLLARY 2.** *If a biconnected 2TSP graph contains an  $s, t$ -nonseparating pair, then it contains a special pair.*

*Proof.* Let  $G$  be a biconnected 2TSP graph that contains one or more  $s, t$ -nonseparating pairs. Let  $T$  denote a compressed decomposition tree for  $G$ . By Lemma 6,  $T$  must contain an  $S$ -node whose children include at least two leaf nodes. Among all such  $S$ -nodes, choose one, say  $x$ , such that no other  $S$ -node in  $T_x$  has two or more leaf nodes as children. Lemma 6 implies that for each nonleaf child  $y$  of  $x$ ,  $T_y$  cannot contain an  $s, t$ -nonseparating pair. Now from among all the leaf nodes that are children of  $x$ , choose two, say  $\hat{e}$  and  $\hat{f}$ , such that no child of  $x$  that occurs between  $\hat{e}$  and  $\hat{f}$  is a leaf. By Lemma 9,  $e$  and  $f$  form a special pair. ■

LEMMA 10. *Let  $G$  be a biconnected 2TSP graph that contains no special pairs. Then  $G$  contains at most one  $s, t$ -separating pair.*

*Proof.* We use contradiction. Suppose  $\{e_1, e_2\}$  and  $\{e_3, e_4\}$  are two distinct  $s, t$ -separating pairs in  $G$ . Assume, without loss of generality, that  $e_1 \neq e_3$  and  $e_2 \neq e_3$ . Let  $G_1$  and  $G_2$  be 2TSP graphs such that  $G$  is the result of their parallel composition. From Tables 1 and 2, we see that any  $s, t$ -separating pair in  $G$  consists of a cut edge in  $G_1$  and a cut edge in  $G_2$ . Thus each of  $e_1, e_2$ , and  $e_3$  is a cut edge in either  $G_1$  or  $G_2$ . Without loss of generality, assume that two of these three edges are in  $G_1$ . But now, by Table 2, these two edges constitute an  $s, t$ -nonseparating pair in  $G$ , contradicting Lemma 9. ■

Recall that it suffices to find three-edge-connected components of biconnected graphs, because each cut edge and cut edge pair in any graph is contained within one of its biconnected components. Lemmas 5 through 9 demonstrate that `remove_non_sep` correctly finds special pairs, adds virtual edges, and updates the decomposition tree to represent the graph left after the edges are removed. By Lemma 10, at most one  $s, t$ -separating pair remains in the graph. This pair, if it exists, is found and removed using `remove_sep`. We omit the analysis of this last step, which at this point is relatively straightforward.

### 3.4. Algorithm test

Algorithm test is the heart of our method. The input to test is a three-edge-connected series-parallel multigraph. In such a graph, suppose  $v$  is a vertex with exactly two neighbors,  $u$  and  $w$ , and suppose there is only one copy of the edge  $uw$ . (Thus there are at least two copies of  $uv$  by three-edge-connectivity.) We say that  $v$  is *pruned* if the multiplicity of  $uv$  is set to 2. Similarly, we say a graph is pruned if each vertex fitting the profile of  $v$  is pruned.

**algorithm test( $G$ )**

input: a three-edge-connected series-parallel multigraph  $G$

output: YES, if  $G$  contains an immersed  $K_4$ , NO otherwise

**begin**

**for** each vertex  $v$  in  $G$  with exactly one neighbor **do**

    delete all but three copies of edges incident on  $v$

**if** any cut point in  $G$  has degree 7 or more

**then** output YES and halt

**for** each biconnected component  $B$  with four or more vertices **do**

**begin**

      prune  $B$

```

    if there is a vertex in  $B$  with degree 5 or more
    then output YES and halt
  end
output NO and halt
end

```

**THEOREM 1.** *Algorithm test runs in  $O(n)$  time on a graph with  $n$  vertices.*

*Proof.* Biconnected components and cut points can be found in linear time using a depth-first search. Operations such as deleting edges incident on vertices with only one neighbor and pruning vertices with two neighbors can be accomplished in constant time. The existence of a vertex with degree 5 or more can be confirmed simply by scanning the list of edges. Thus the theorem holds. ■

### 3.5. The Correctness of test

The correctness of test relies on a number of lemmas, which follow. Before proceeding, we make a few useful observations.

**OBSERVATION 1.** *If  $H'$  is immersed in  $H$ , and if  $M'$  is a  $K_4$  model in  $H'$ , then in  $H$  there is a  $K_4$  model  $M$  with the same corners as  $M'$ .*

Observation 1 follows from noting that edges in  $H'$  map to edge-disjoint paths in  $H$ , and that in  $H$  a suitable  $K_4$  model can be found merely by replacing the edges of  $M'$  with their image paths in  $H$ .

Observation 2 is a well-known property of series-parallel graphs (see [13], for example, for a proof).

**OBSERVATION 2.** *Every biconnected series-parallel multigraph with four or more vertices contains at least two nonadjacent vertices with exactly two neighbors.*

Suppose  $v$  is a vertex with exactly two neighbors,  $x$  and  $y$ . We say that  $v$  is *shorted* if we lift all pairs of edges  $vx$  and  $vy$  and if we delete any remaining edges incident on  $v$  along with  $v$  itself.

**OBSERVATION 3.** *Shorting preserves biconnectivity, three-edge-connectivity, and series-parallelness.*

Observation 3 holds because shorting a vertex does not change the number of vertex-disjoint or edge-disjoint paths between any pair of remaining vertices.

**OBSERVATION 4.** *A biconnected component of a three-edge-connected graph is three-edge-connected.*

Observation 4 follows from noting that edge-disjoint paths may as well be made simple and that, whenever a pair of vertices lies in the same biconnected component, all vertices along simple paths connecting them in the original graph must also lie in this component.

The next lemma justifies the first step in test, in which edges incident on vertices with only one neighbor have their multiplicity reduced to 3.

**LEMMA 11.** *Let  $G$  denote a graph in which a vertex,  $v$ , has exactly one neighbor,  $w$ . Let  $G'$  be obtained from  $G$  by deleting all but three copies of the edge  $vw$ . Then  $K_4$  is immersed in  $G$  if and only if  $K_4$  is immersed in  $G'$ .*

*Proof.* If  $K_4$  is immersed in  $G$ , then  $G$  contains a  $K_4$  model whose edge images are simple paths. Because  $v$  cannot be an intermediate vertex in a simple path, at most three copies of the edge  $vw$  are needed. Thus  $K_4$  is also immersed in  $G'$ . If  $K_4$  is not immersed in  $G$ , then neither is it immersed in  $G'$  because  $G'$  is a subgraph of  $G$ . ■

Under certain conditions, it is possible to detect an immersed  $K_4$  by checking whether the graph in Fig. 3, henceforth termed graph  $M$ , is immersed in the input graph. The next four lemmas are useful in finding  $M$ -models.

**LEMMA 12.** *Let  $G$  be three-edge-connected, with noncut point vertices  $u$  and  $v$ . Let  $w \notin \{u, v\}$  denote a vertex in  $G$ . Then there exist three mutually edge-disjoint paths, each beginning with  $w$  and ending with either  $u$  or  $v$ , such that at most two of these paths contain  $u$ , at most two contain  $v$ , and none contains both  $u$  and  $v$ .*

*Proof.* The paths we seek to identify are illustrated in Fig. 4, with dashed lines denoting edge-disjoint paths that do not contain  $u$  or  $v$  as an intermediate vertex. Consider three mutually edge-disjoint paths  $P_1$ ,  $P_2$ , and  $P_3$ , each from  $w$  to  $\{u, v\}$ . These paths exist because  $G$  is three-edge-connected. Assume all three contain, say,  $u$ . Hence all three may as well be simple and end at  $u$ . Consider now some path  $P$  between  $w$  and  $v$  that does not contain  $u$  (such a path exists because  $u$  is not a cut point).  $P$  may contain vertices and edges in  $P_1$ ,  $P_2$ , and  $P_3$ . Let  $y$  be the last vertex in  $P$  (counting from  $w$ ) that is also in  $P_1$  or  $P_2$  or  $P_3$ . Without loss of

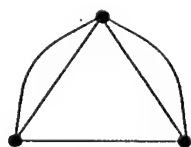


FIG. 3. The graph of  $M$ .

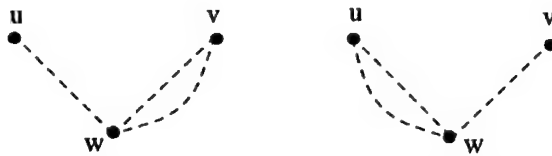


FIG. 4. Edge-disjoint paths in a three-edge-connected graph.

generality, assume  $y$  is in  $P_1$ . We can construct a path  $P'$  from  $w$  to  $v$ , by taking  $P_1$  until we reach  $y$ , and using  $P$  from there on. Thus  $P'$ ,  $P_2$ , and  $P_3$  are the desired edge-disjoint paths, with  $P'$  not containing  $u$ . ■

**LEMMA 13.** *Let  $G$  be three-edge-connected. Let  $v$  denote a noncut point vertex in  $G$  with degree at least 4, let  $u$  and  $w$  be neighbors of  $v$ , and suppose  $uv$  has multiplicity at least 2. Then  $G$  contains an  $M$  model, with corners  $u$ ,  $v$ , and  $w$ , and with  $v$  the image of  $M$ 's degree-4 vertex.*

*Proof.* We restrict our attention to  $G'$ , the biconnected component of  $G$  containing  $v$ . ( $G'$  is three-edge-connected by Observation 4. Because  $v$  is not a cut point, its neighborhood is unchanged in  $G'$ .) From Lemma 12, we know that there are three mutually edge-disjoint paths from  $w$  to  $\{u, v\}$  such that at most two of these paths contain  $u$  and at most two of these paths contain  $v$ . One of these paths is the edge  $wv$ . If one of the other paths contains  $v$  as well, the lemma holds. So suppose neither contains  $v$ . This situation is illustrated in Fig. 5a. To complete an  $M$  model, we must find an edge-disjoint path  $[vw]$ . If  $uv$  has multiplicity 3 or more, we can construct this path by combining one of the edges  $vu$  and one of the paths  $[uw]$ . So assume  $uv$  has multiplicity 2, and let  $x$  denote a neighbor of  $v$  other than  $u$  or  $w$ . Because  $G'$  is biconnected, there is a path  $[xw]$  that does not contain any of the edges incident on  $v$ . Let  $y$  denote the first vertex on this path (counting from  $x$ ) common to either of the two paths  $[uw]$ . We can combine the edge  $ux$  with the paths  $[xy]$  and  $[yw]$  to get the desired path  $[vw]$ , as is clear from Fig. 5b. ■



FIG. 5. Graphs used in the proof of Lemma 13.

LEMMA 14. *Let  $G$  be three-edge-connected, with at least three vertices. Let  $v$  denote a vertex in  $G$  with only one neighbor  $u$ , and let  $w \neq v$  denote a neighbor of  $u$ . Suppose  $v$  has degree at least 4. Then  $G$  contains an  $M$  model, with corners  $u, v$ , and  $w$ , and with  $v$  the image of  $M$ 's degree-4 vertex.*

*Proof.* The graph depicted in Fig. 6 is immersed in  $G$ . A satisfactory model of  $M$  can be obtained by lifting two pairs of edges in this graph. ■

LEMMA 15. *Let  $G$  be three-edge-connected and series-parallel, with at least three vertices. Let  $v$  denote a vertex in  $G$  with degree at least 4. Then  $G$  contains an  $M$  model in which  $v$  is the image of  $M$ 's degree-4 vertex.*

*Proof.* If  $v$  has only one neighbor, then the model exists by Lemma 14. Otherwise let  $u$  and  $w$  denote arbitrary neighbors of  $v$ . If  $v$  is a cut point, then the graph in Fig. 7 is immersed in  $G$ , satisfying the statement of the lemma. So suppose  $v$  is not a cut point. If any vertex in  $G$  is adjacent to  $v$  by two or more edges, then Lemma 13 applies, and the model exists. So suppose there are no edges of multiplicity greater than 1 incident on  $v$ . Then we may force this condition by iteratively deleting vertices with only one neighbor and shorting vertices with only two neighbors. Neither operation changes the degree of  $v$ , or affects the three-edge-connectivity and series-parallelness of  $G$ . Thus, by the time the order of  $G$  is reduced to 3 (or before), some edge incident on  $v$  must have multiplicity 2 or more. Lemma 13 and Observation 1 then imply that the lemma holds. ■

The preceding lemmas enable us to detect  $K_4$  models that span cut points.

LEMMA 16. *Let  $G$  be three-edge-connected and series-parallel, and let all vertices in  $G$  with exactly one neighbor have degree 3. Suppose  $G$  has a cut point  $v$  with degree 7 or more. Then  $K_4$  is immersed in  $G$ .*

*Proof.* Let  $C_1, \dots, C_k$  denote the connected components of  $G - \{v\}$ . Let  $A_i$  denote  $C_i$  augmented with a copy of  $v$  and the edges it induces. Each  $A_i$  is three-edge-connected, and thus contains a model of the triple-edge shown in Fig. 8a, with any pair of vertices serving as the corners. Without loss of generality, assume  $A_1$  contains the least number of edges incident on  $v$ , and let  $H$  denote  $G - C_1$ . It follows that  $v$  has degree 4 or more in  $H$  and that  $H$  has at least three vertices. Thus, by Lemma 15, there is an  $M$  model in  $H$  with  $v$  the image of the degree-4

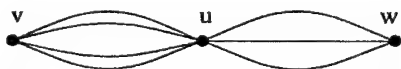


FIG. 6. Graph used in the proof of Lemma 14.

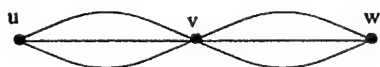


FIG. 7. Graph used in the proof of Lemma 15.

vertex in  $M$ . This  $M$  model can be combined with a model of the triple-edge in  $A_1$  to form in  $G$  a model of the graph shown in Fig. 8b, which contains an immersed  $K_4$ . ■

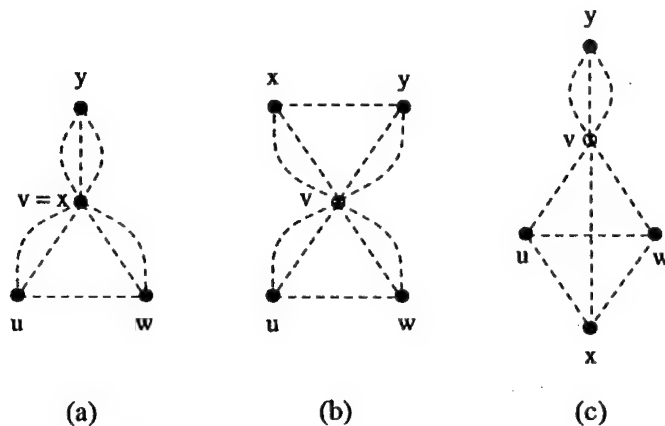
$K_4$  models that span cut points may exist even if no cut points have degree 7 or more. Nevertheless, we can restrict our search to biconnected components, as we show in the following text.

**LEMMA 17.** *Suppose  $G$  has no cut point with degree exceeding 6. Then  $K_4$  is immersed in  $G$  if and only if  $K_4$  is immersed in a biconnected component of  $G$ .*

*Proof.* If a biconnected component of  $G$  contains  $K_4$ , then so does  $G$ , because a biconnected component is a subgraph. To prove the converse, consider a  $K_4$  model in  $G$  with the  $K_4$  edges mapped to simple paths. Let  $u, w, x$ , and  $y$  denote the corners of this model, and suppose there is a cut point  $v$  that separates them. We know that  $v$  cannot be one of the corners, else it would need degree 7 or more (see Fig. 9a).  $v$  cannot separate two corners from the others, else it would need degree 8 or more (see Fig. 9b). So it must be that  $v$  separates just one corner, say  $y$ , from the others (see Fig. 9c). Thus the edge-disjoint paths  $[uy]$ ,  $[wy]$ , and  $[xy]$  all pass through  $v$ , and we can construct another  $K_4$  model in which  $v$  replaces  $y$  as a corner. By iterating this replacement, we eventually get a  $K_4$  model all of whose corners (and paths) are in the same biconnected component. ■



FIG. 8. Graphs used in the proof of Lemma 16.

FIG. 9. Models of  $K_4$  that span a cut point.

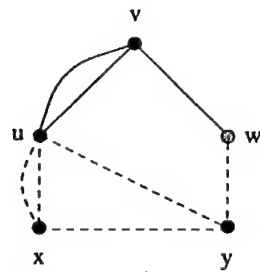
The remaining lemmas in this section deal with detecting an immersed  $K_4$  in a biconnected, three-edge-connected, series-parallel graph. First we show that such a graph must have at least one vertex of degree 5 or more if it is to contain an immersed  $K_4$ . Then we present a series of lemmas that lead up to a proof of the converse; that is, if even one vertex has degree 5 or more, then an immersed  $K_4$  exists.

**LEMMA 18.** *Let  $v$  denote a vertex with exactly two neighbors,  $u$  and  $w$ , and suppose the edge  $vw$  has multiplicity 1. Then  $u$  and  $v$  can be corners of a given  $K_4$  model only if  $\text{degree}(u) \geq \text{degree}(v) + 2$ .*

*Proof.* Let  $x$  and  $y$  denote the other corners of this model. Paths  $[ux]$  and  $[uy]$  need not contain  $uv$ . Either  $[ux]$  or  $[vy]$  has to pass through  $u$ . Thus at least three edges are incident on  $u$  in addition to the copies of  $uv$  (see Fig. 10), and the lemma follows. ■

**LEMMA 19.** *If  $G$  is series-parallel and of maximum degree 4, then  $K_4$  is not immersed in  $G$ .*

*Proof.* Suppose otherwise, and let  $H$  denote a minimal counterexample.  $H$  must be three-edge-connected by Lemma 1.  $H$  must also be biconnected, because a cut point in a three-edge-connected graph has degree at least 6. Thus, by Observation 2,  $H$  contains a vertex,  $v$ , with exactly two neighbors,  $u$  and  $w$ . It must be that  $v$  is needed as a corner in every  $K_4$  model, else we can short it, contradicting minimality. So  $v$  has degree 3 and we assume, without loss of generality, that  $uv$  has multiplicity 2,  $vw$  has multiplicity 1. We now fix the remaining corners of some  $K_4$  model. Vertex  $u$  cannot be one of these corners, by Lemma 18. But now it

FIG. 10. A model of  $K_4$  with corners  $u, v, x$ , and  $y$ .

is easy to see that  $u$  can replace  $v$  in this model, contradicting the fact that  $v$  must be a corner. ■

We henceforth use the term *candidate graph* to denote a biconnected, three-edge-connected, series-parallel multigraph with four or more vertices.

**LEMMA 20.** *In a candidate graph,  $G$ , suppose vertex  $v$  has exactly two neighbors,  $u$  and  $w$ , and suppose the multiplicity of  $uv$  is greater than the multiplicity of  $vw$ . If  $\text{degree}(u) - \text{degree}(v) \geq 2$ , then  $K_4$  is immersed in  $G$ .*

*Proof.* Suppose otherwise, and let  $H$  denote a minimal counterexample. Let  $x \neq v$  denote another vertex with exactly two neighbors. The edge  $xu$  must exist and have multiplicity 2 or more, else we can short  $x$  without affecting the degree of  $u$  or  $v$ , thus contradicting the minimality of  $G$ . Consider the effect of shorting  $v$ , producing the graph  $H'$ . Because  $u$  has degree at least 4 in  $H'$ , we know from Lemma 13 that the  $M$  model illustrated in Fig. 11a is immersed in  $H'$ . But this means that the graph shown in Fig. 11b, which contains  $K_4$ , is immersed in  $H$ , thereby contradicting the assumption that  $H$  is a counterexample. ■

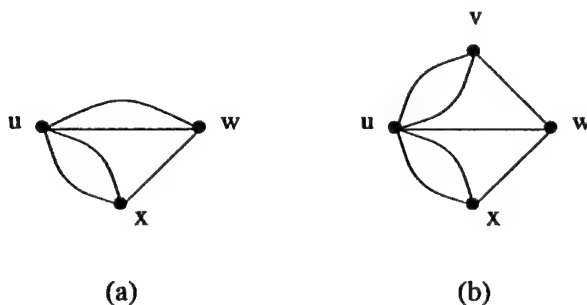


FIG. 11. Graphs used in the proofs of Lemmas 20 and 22.

Recall pruning, as defined in Section 3.4.

**LEMMA 21.** *In a candidate graph,  $G$ , suppose vertex  $v$  has exactly two neighbors,  $u$  and  $w$ , and suppose  $vw$  has multiplicity 1. Letting  $G'$  denote the graph resulting from pruning  $v$ ,  $K_4$  is immersed in  $G$  if and only if it is immersed in  $G'$ .*

*Proof.* If  $K_4$  is immersed in  $G'$ , then it is immersed in  $G$  as well, because  $G' \subseteq G$ . Suppose  $K_4$  is immersed in  $G$ . If  $G$  contains a  $K_4$  model in which  $v$  is not a corner, then so does  $G'$ , because pruning is irrelevant (at most one of the images of the  $K_4$  edges in this model can pass through  $v$ ). So suppose  $v$  is a corner in every  $K_4$  model in  $G$ . Vertex  $u$  must also be a corner in all these models, else we could replace  $v$  with  $u$ , forming a model in which  $v$  is not a corner. Now, by Lemma 18,  $u$  has degree at least 2 more than  $v$ , a property unchanged by pruning. Thus, by Lemma 20,  $K_4$  is immersed in  $G'$ . ■

**LEMMA 22.** *In a pruned candidate graph,  $G$ , suppose vertex  $v$  has exactly two neighbors,  $u$  and  $w$ , and suppose  $uv$  has multiplicity at least 3. Then there is a  $K_4$  model in  $G$  with corners  $u$ ,  $v$ ,  $w$ , and  $x$ , where  $x \notin \{v, w\}$  is a neighbor of  $u$ .*

*Proof.* Vertex  $u$  must have some neighbor not in  $\{v, w\}$  to play the role of  $x$ , else  $w$  would be a cut point, contradicting the biconnectivity of  $G$ . Because  $G$  is pruned, edge  $vw$  must have multiplicity at least 2, and the degree of  $u$  must be at least two more than the multiplicity of  $uv$ . Thus in  $G'$ , the graph that results from shorting  $v$ , the degree of  $u$  is at least 4. We conclude from Lemma 13 that the  $M$  model illustrated in Fig. 11a is immersed in  $G'$ , and the graph shown in Fig. 11b, which contains  $K_4$ , is immersed in  $G$ . ■

**LEMMA 23.** *In a candidate graph,  $G$ , suppose vertex  $v$  has exactly two neighbors,  $u$  and  $w$ , suppose  $uv$  and  $vw$  each have multiplicity at least 2, and suppose  $uw$  exists. Then there is a  $K_4$  model in  $G$  with corners  $u$ ,  $v$ ,  $w$ , and  $x$ , where  $x \notin \{v, w\}$  is a neighbor of  $u$ .*

*Proof.* As in the last lemma, such an  $x$  must exist. We apply Lemma 12, with  $w$  playing the role of  $v$  and  $x$  playing the role of  $w$ . Thus at least one of the graphs shown in Fig. 12, both of which contain  $K_4$ , is immersed in  $G$ . ■

**LEMMA 24.** *Let  $G$  denote a pruned candidate graph.  $K_4$  is immersed in  $G$  if and only if  $G$  has a vertex of degree 5 or more.*

*Proof.* We know from Lemma 19 that a candidate graph of maximum degree 4 contains no  $K_4$ . To prove the converse, we proceed by contradiction and we assume  $H$  denotes a minimal pruned candidate graph, with at

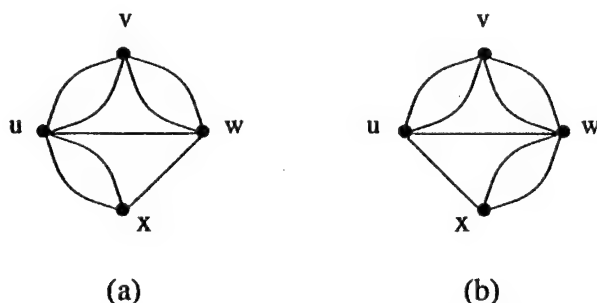


FIG. 12. Graphs used in the proof of Lemma 23.

least one vertex of degree 5 or more, but with no immersed  $K_4$ . We observe that  $H$  must contain more than four vertices. Otherwise, let  $a$  denote a vertex in  $H$  with degree 5 or more. If  $a$  has only two neighbors, then the graph of Fig. 13a, which contains  $K_4$ , is a subgraph of  $H$ , contradicting our assumption that  $H$  contains no  $K_4$ . If  $a$  has three neighbors, then the graph of Fig. 13b, which also contains  $K_4$ , is a subgraph of  $H$ , leading once more to a contradiction. Thus  $H$  has at least five vertices, a necessary property because we will use shorting to contradict minimality, and a candidate graph requires at least four vertices. Let  $v$  denote a vertex in  $H$  with exactly two neighbors,  $u$  and  $w$ , and assume the multiplicity of  $uv$  is at least that of  $vw$ . Lemma 22 guarantees that  $v$  cannot have degree 5 or more. If  $v$  has degree 4, Lemma 23 and the fact that  $H$  is pruned ensure that  $uw$  does not exist. But now we can short  $v$ , obtaining a pruned candidate graph that contradicts minimality. So  $v$  must have degree 3 and, by Lemma 20,  $u$  has degree 4 or less. Biconnectivity requires that at most one copy of  $uw$  exists. But now we can again short  $v$  to obtain a pruned candidate graph, contradicting the presumed minimality of  $H$ . ■

This completes the proof of the correctness of test. The work of the last two sections provides the proof of the following principal result.

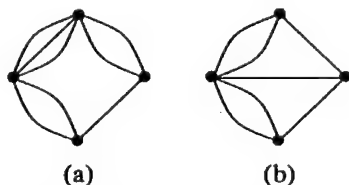


FIG. 13. Pruned four-vertex graphs that contain a vertex of degree 5.

**THEOREM 2.** *Algorithms decompose, components and test correctly decide in linear time whether  $K_4$  is immersed in an arbitrary input graph.*

#### 4. FINDING A MODEL

Once the presence of  $K_4$  has been detected in a graph, our method to identify a  $K_4$  model proceeds in two steps. Algorithm corners is first invoked to modify the input graph until an appropriate set of corners is isolated. Then algorithm paths is used to find the  $K_4$  edge images.

##### 4.1. Algorithm corners

Algorithm corners marks vertices in the input graph as part of the corner-finding process. All vertices are assumed to be unmarked initially. Algorithm corners also maintains a list for every copy of every edge, to store the sequence of edges that may have been eliminated by shorting. Each list is assumed to contain only the edge itself initially.

**algorithm corners( $G$ )**

input: a three-edge-connected series-parallel multigraph  $G$  containing an immersed  $K_4$

output: the four corners of a  $K_4$  model in  $G$

**begin**

**for** each vertex with only one neighbor

    delete all but three copies of its incident edge

**if**  $G$  has a cut point  $v$  of degree 7 or more

**then**  $u, v, w, x := \text{spanning-corners}(G, v)$

**else begin**

      prune each biconnected component of  $G$

$C :=$  a biconnected component with four or more vertices of which at least one has degree 5 or more

$u, v, w, x := \text{biconnected-corners}(C)$

**end**

  output  $u, v, w, x$

**end**

We address the correctness of corners. Suppose  $G$  contains a cut point  $v$  of degree at least 7, after redundant edges incident on vertices with only one neighbor are deleted. In this case, we use algorithm spanning-corners to locate the corners.

**algorithm spanning-corners( $G, v$ )**

input: a three-edge-connected series-parallel multigraph  $G$  in which each vertex with exactly one neighbor has degree 3 and a cut point  $v$  with degree 7 or more

output: the four corners of a  $K_4$  model in  $G$

begin

  if  $G - \{v\}$  has three or more connected components

  then /\* Case 1a \*/

$u, w, x :=$  neighbors of  $v$  in  $G$ , each in a different connected component of  $G - \{v\}$

  else begin /\* Case 1b \*/

$C_1, C_2 :=$  the connected components of  $G - \{v\}$

$A_1 := C_1$  augmented with  $v$  and the edges it induces

$A_2 := C_2$  augmented with  $v$  and the edges it induces

    if  $v$  has degree 4 or more in  $A_1$

      then  $A := A_1$  and  $B := A_2$

      else  $A := A_2$  and  $B := A_1$

    while  $v$  induces no edge of multiplicity 2 or more in  $A$

      if there is a vertex in  $A$  with only one neighbor

        then delete this vertex and its incident edges

        else short some vertex in  $A$  with only two neighbors

$u :=$  some vertex in  $A$  such that  $uv$  has multiplicity at least 2

    if  $v$  has no neighbor other than  $u$  in  $A$

      then  $w :=$  any neighbor of  $u$  in  $A$  other than  $v$

      else  $w :=$  any neighbor of  $v$  in  $A$  other than  $u$

$x :=$  any neighbor of  $v$  in  $B$

  end

  output  $u, v, w, x$

end

If  $G - \{v\}$  has three or more connected components (Case 1a), it follows from the three-edge-connectivity of  $G$  that a model of the star graph shown in Fig. 1 exists in  $G$ , with  $v$  playing the role of the central vertex. Any three vertices in  $G - \{v\}$  can serve as the remaining corners, as long as no two of them are in the same connected component of  $G - \{v\}$ . Thus the vertices returned are the corners of a  $K_4$  model.

If  $G - \{v\}$  has only two components (Case 1b), then we apply Lemmas 13 and 14. If  $v$  has only one neighbor  $u$  in the augmented component  $A$  (see algorithm spanning-corners), then by Lemma 14, we may choose  $v$ ,  $u$ , and an arbitrary neighbor  $w$  of  $u$  as the corners of an  $M$  model. If  $v$  has two or more neighbors, then the corners of  $M$  can be found using Lemma 13 as long as an edge of multiplicity 2 or more is incident on  $v$  in  $A$  (note that  $v$  cannot be a cut point in  $A$ ). As observed in the proof of Lemma 15, if this condition is not initially satisfied, it can easily be forced by deleting and shorting vertices. The corners of the  $M$  model serve as three of the four corners of a  $K_4$  model. The vertex  $x$  chosen as the fourth corner belongs to the connected component of  $G - \{v\}$  that does

not contain the first three corners. By the three-edge-connectivity of  $G$ , a model of the triple-edge exists in  $G$ , with  $v$  and  $x$  as corners. Combining this model with the  $M$  model, we obtain a model of the graph shown in Fig. 8b. Lifting two pairs of edges in this graph gives us  $K_4$ .

If  $G$  contains no cut point of degree 7 or more, then biconnected-corners is invoked on some pruned biconnected component of  $G$  that contains at least one vertex of degree 5 or more. Lemma 24 implies that such a biconnected component exists, and moreover, that it contains an immersed  $K_4$ . This component must contain a vertex with exactly two neighbors (Observation 2). In this event, we employ Lemmas 22 and 23, plus Lemma 25, which follows.

**algorithm** biconnected-corners( $G$ )

input: a three-edge-connected biconnected pruned series-parallel multi-graph  $G$  with at least four vertices and with at least one vertex of degree 4 or more

output: the four corners of a  $K_4$  model in  $G$

**begin**

**while**  $x$  has not been assigned a value **do**

**begin**

$v :=$  an unmarked vertex with exactly two neighbors

$u, w :=$  the neighbors of  $v$ , with the multiplicity of  $uw$  at least that of  $vw$

**if**  $v$  has degree at least 5, or  $v$  has degree 4 and  $uw$  exists

**then** /\* Case 2a \*/

$x :=$  any neighbor of  $u$  besides  $v$  or  $w$

**else if**  $v$  or  $u$  has degree 4

**then** short  $v$

**else if**  $uw$  exists

**then** /\* Case 2b \*/

$x :=$  any neighbor of  $u$  other than  $v$  or  $w$

**else if** there is an edge  $ua$ ,  $a \neq v$ , of multiplicity 2 or more

**then** /\* Case 2c \*/

$x := a$

**else if** there are two vertices of degree 5 or more

**then** short  $v$  **else** mark  $v$

**end**

  output:  $u, v, w, x$

**end**

**LEMMA 25.** *In a candidate graph,  $G$ , suppose vertex  $v$  has exactly two neighbors,  $u$  and  $w$ , and suppose  $uw$  has multiplicity 2,  $vw$  has multiplicity 1, and  $u$  has degree at least 5. Let  $x$  denote a neighbor of  $u$  other than  $v$  or  $w$ . If*

$ux$  has multiplicity at least 2 or  $uw$  exists, then there is a  $K_4$  model in  $G$  with  $u, v, w$ , and  $x$  as corners.

*Proof.* In  $G'$ , the graph resulting from shorting  $v, u$  has degree at least 4, and either  $ux$  has multiplicity at least 2 or  $uw$  now does. Then by Lemma 13, there is an  $M$  model in  $G'$  with corners  $u, w$ , and  $x$ , and with  $u$  the image of  $M$ 's degree-4 vertex. Thus the graph in Fig. 11b, which contains the desired  $K_4$  model, is immersed in  $G$ . ■

Let  $v$  be defined as an algorithm biconnected-corners. The corners of an immersed  $K_4$  can be found if one of the following conditions holds:

- either  $v$  has degree at least 5 or  $v$  has degree 4 and edge  $uw$  exists (Case 2a). Lemma 22 applies in the former situation, and Lemma 23 in the latter situation.
- $v$  has degree 3,  $u$  has degree 5 or more, and either edge  $uw$  exists or there is an edge  $ua$ ,  $a \neq v$ , of multiplicity 2 or more (Cases 2b and c). Lemma 25 applies.

If an immersed  $K_4$  cannot yet be identified, then a vertex,  $v$ , with exactly two neighbors is shorted as long as the resulting graph retains at least one vertex of degree at least 5. Accordingly, if one of  $v$ 's neighbors,  $u$ , is the only vertex of degree at least 5,  $uv$  has multiplicity 2, and all other edges incident on  $u$  are simple, then  $v$  cannot be shorted. It suffices in this case to mark  $v$  as having been visited, because at most one vertex can be so marked and another candidate for  $v$  is always available. Finally, we note that continued shorting will never result in a graph with fewer than four vertices. This is because, as observed in the proof of Lemma 24, one of the two graphs shown in Fig. 13 must be a subgraph of any candidate graph that contains exactly four vertices of which at least one has degree 5 or more. Because one of Cases 2a–c apply to any vertex with exactly two neighbors in these two graphs, corners are identified at this point without any vertex being marked or shorted. Of course, it is possible that corners are found before the graph is reduced to four vertices.

Biconnected components and cut points can be found (using a depth-first search) and vertices pruned, in linear time. It takes only linear time to check whether a biconnected component contains a vertex of degree 5 or more. In each iteration of spanning-corners, some vertex with two or fewer neighbors is deleted or shorted. In each iteration of biconnected-corners, some vertex with exactly two neighbors is shorted or marked. Vertices with two or fewer neighbors can be maintained using a queue. Deleting, shorting, or marking such a vertex, and updating the queue and the appropriate edge list require only a constant number of steps. Thus both spanning-corners and biconnected-corners run in linear time, and hence, so does corners.

4.2. *Algorithm paths*

Algorithm *paths* uses the property that  $k$  edge-disjoint paths exist between a pair of vertices if and only if a network flow of value  $k$  is possible between them.

**algorithm** *paths*( $G, s, t_1, \dots, t_k$ )

input: a multigraph  $G$  and distinguished vertices  $s, t_1, \dots, t_k$

output: edge-disjoint paths  $p_1, \dots, p_k$ , with  $p_i$  connecting  $s$  to  $t_i$ , if such paths exist

**begin**

$G' :=$  the edge-weighted digraph obtained by replacing each edge  $uv$  of multiplicity  $m$  with the directed edges  $(u, v)$  and  $(v, u)$ , each of capacity  $m$

add to  $G'$  a vertex  $t$  and the edges  $(t_1, t), \dots, (t_k, t)$ , each of capacity 1

find a flow of value  $k$  from  $s$  to  $t$ , if such a flow exists

**if** there is such a flow **then**

**begin**

**for** each edge  $(u, v)$  in  $G'$  **do**

**if** both  $(u, v)$  and  $(v, u)$  have positive flow values

**then**  $\text{flow}((u, v)) := \max(0, \text{flow}((u, v)) - \text{flow}((v, u)))$  and

$\text{flow}((v, u)) := \max(0, \text{flow}((v, u)) - \text{flow}((u, v)))$

discard from  $G'$  any edge without a positive flow

**for**  $i = 1$  to  $k$  **do**

**begin**

$p'_i :=$  a path in  $G'$  from  $s$  to  $t_i$

$p_i :=$  the corresponding path in  $G$

decrement in  $G'$  the flow along each edge in  $p'_i$  by one

delete from  $G$  one copy of each edge in  $p_i$

**end**

output  $p_1, \dots, p_k$

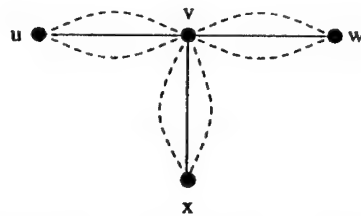
**end**

**end**

We address the correctness and use of *paths*. In the following figures, paths that are mere edges are shown as solid lines. These edges are temporarily deleted so that paths can be employed to find additional paths with multiple edges, depicted with dashed lines. We consider various cases.

*Case 1a.* The  $K_4$  model spans a cut point  $v$ , and  $G - \{v\}$  has three or more connected components.

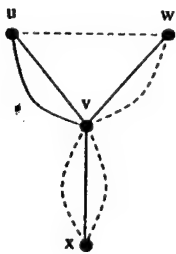
Corners  $u, w$ , and  $x$  are in different connected components of  $G - \{v\}$ , and each corner is adjacent to  $v$  in  $G$ . Two paths need to be found between each corner and  $v$ , to complete a model of the star graph in Fig. 1 and hence a  $K_4$  model. So three calls are made to *paths*, each with  $v$

FIG. 14. Paths to be found if  $G - \{v\}$  has three or more connected components.

playing the role of  $s$  and  $k$  set to 2. See Fig. 14. The first call uses  $u = t_1 = t_2$ ; the second call uses  $w = t_1 = t_2$ ; the third call uses  $x = t_1 = t_2$ .

*Case 1b.* The  $K_4$  model spans a cut point  $v$ , and  $G - \{v\}$  has only two connected components.

See Fig. 15. Figure 15a depicts the case when the biconnected component of  $G$  containing both  $u$  and  $v$  has at least one corner. In this case, corners  $u$ ,  $w$ , and  $x$  are all adjacent to  $v$ . Moreover, edge  $uv$  has multiplicity 2 or more. Note that  $v$  separates  $x$  from  $u$  and  $w$ . To complete a model of the graph shown in Fig. 8b, two paths between  $v$  and  $x$  in the biconnected component containing  $v$  and  $x$ , and two paths between  $w$  and  $\{u, v\}$  in the biconnected component containing  $u$ ,  $v$ , and  $w$  must be found. Thus two calls to paths are required. The first call uses  $w = s$ ,  $u = t_1$ , and  $v = t_2$ ; the second call uses  $v = s$  and  $w = t_1 = t_2$ . Figure 15b depicts the case when  $u$  and  $v$  are in a biconnected component by themselves. Corner  $x$  is adjacent to  $v$  and corner  $w$  is adjacent to  $u$ . Again, two calls to paths are required. The first call uses  $u = s$ ,  $w = t_1 = t_2$ ; the second call uses  $v = s$  and  $x = t_1 = t_2$ .

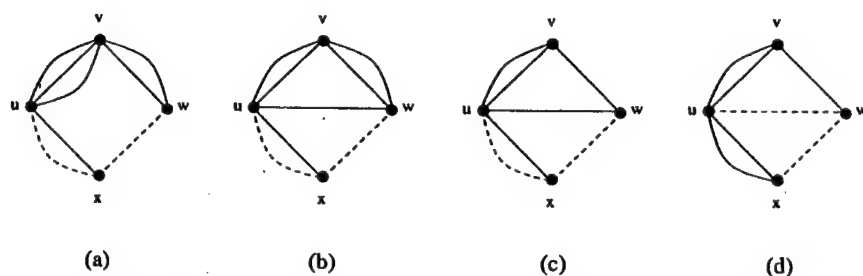


(a)



(b)

FIG. 15. Paths to be found if  $G - \{v\}$  has only two connected components.

FIG. 16. Paths to be found if the  $K_4$  model lies in a biconnected component.

Cases 2a-c. The  $K_4$  model is in a single biconnected component. See Fig. 16. Case 2a of algorithm corners is illustrated in Figs. 16a and b. Cases 2b and c are illustrated in Figs. 16c and d, respectively. In each case, one call to paths, with  $k$  set to two, suffices.

Recall that the input to paths has at most a linear number of edges and no more than four copies of any edge. Thus it takes only linear time to construct  $G'$  and to read off paths (using, for example, a breadth-first search) after a flow of value  $k$  has been found. The running time of paths is therefore dominated by the algorithm for finding network flows. So we employ a flow method such as Ford-Fulkerson [6], which runs in linear time as long as  $k$  is bounded by an integer constant and all edge-capacities are integers, as is the case here.

In summary, to find a  $K_4$  model we invoke corners once and paths at most three times. Because both corners and paths are linear-time algorithms, the entire model-finding process is accomplished in linear time.

**THEOREM 3.** *If  $K_4$  is immersed in an arbitrary series-parallel graph, algorithms corners and paths correctly isolate a  $K_4$  model in linear time.*

## 5. DISCUSSION

We have presented linear-time methods to detect if  $K_4$  is immersed in an input graph, and to isolate a  $K_4$  model if any exist. We implemented our algorithms in C and we ran them on a SUN SPARCstation 20. Experiments on randomly generated graphs indicate that our algorithms are practical, taking only seconds to process graphs with thousands of vertices. The running time of the detection algorithm is affected mainly by the size of the input graph. One might suspect that the distribution of edges over vertices might also have an effect, but we sampled several edge-probability distributions and we could find no noticeable differences.

On the other hand, the model-finding algorithm does appear to take slightly longer on graphs in which we have forced corners to be connected only by long paths. Even on such contrived instances, finding a model takes no more than twice the average time for random graphs of similar size.

We note that the detection algorithm can be efficiently parallelized. Biconnected components and cut points can be found in  $O(\log n)$  time on a CRCW PRAM with  $O((m+n)\alpha(m,n)/\log n)$  processors [10], where  $\alpha(m,n)$  denotes the inverse of Ackermann's function. Deciding whether a graph is series-parallel can be done in  $O(\log^2 n + \log m)$  time with  $O(m+n)$  processors [11]. Recalling that graphs of interest have at most a linear number of edges, a parallel version of decompose thus needs at most  $O(\log^2 n)$  time with  $O(n)$  processors. The triconnected components algorithm of [10], modified slightly to find three-edge-connected components [18], yields a parallel version of components that runs in  $O(\log n)$  time with  $O(n \log \log n / \log n)$  processors. It is straightforward to parallelize test so that it takes constant time with  $O(n)$  processors. Besides finding cut points, which are available from components, the only operations in test are pruning vertices with just two neighbors and deleting edges incident on vertices with only one neighbor. Both can be accomplished in constant time with  $O(n)$  processors. Thus it is possible to determine whether a graph has an immersed  $K_4$  in  $O(\log^2 n)$  time with  $O(n)$  processors on the CRCW PRAM model. We did not implement this scheme because many of the algorithms mentioned are highly impractical. The problem of devising an efficient parallel model-finding method remains open.

Fast immersion tests are of interest in their own right. In practice, they also have potential as indicators of graph width metrics. To illustrate, we return to the cutwidth problem, which has appeared in a wide variety of VLSI applications (see, as examples, [7, 12]). Deciding whether a graph has small cutwidth is an important part of many layout processes. Graphs representing circuits are frequently series-parallel. More generally, they tend to be sparse, with at most a linear number of edges, and of bounded degree due to limitations on porting and fan-in/out. Integer weights are used to model multiple edges in these applications, just as we have used them here. The presence of an immersed  $K_4$  in such a graph guarantees that it cannot have cutwidth 3. The absence of  $K_4$ , however, merely approximates its cutwidth at 3. In particular, such an absence says nothing at all about how to find a layout of width 3 even if many should exist. To solve this problem, our algorithms can be used in conjunction with previously studied "self-reduction" techniques [1, 9] to search for a layout in  $O(n^2)$  time.

Many other combinatorial problems may benefit from fast immersion tests. For example, a variety of *load factor* [8] problems can be decided by a finite battery of immersion tests, including  $K_4$ . A problem indirectly approachable with this method is graph bisection. Bounded cutwidth is a sufficient, but not a necessary, condition for bounded bisection width. For problems such as these, there is interest in devising fast tests for other key graphs [16, 17].

### ACKNOWLEDGMENT

We thank an anonymous referee, whose careful review of our original submission helped us to clarify the presentation of these results.

### REFERENCES

1. D. J. Brown, M. R. Fellows, and M. A. Langston, Polynomial-time self-reducibility: Theoretical motivations and practical results, *Internat. J. Comput. Math.* **31** (1989), 1–9.
2. H. L. Bodlaender, A linear time algorithm for finding tree-decompositions of small treewidth, in "Proceedings Twenty-Fifth Annual ACM Symposium on Theory of Computing," 1993, pp. 226–234.
3. G. A. Dirac, In abstrakten graphen vorhandene vollstandige 4-graphen und ihre unterteilungen, *Math. Nachr.* **22** (1960), 61–85.
4. R. J. Duffin, Topology of series-parallel networks, *J. Math. Anal. Appl.* **10** (1965), 303–318.
5. S. Even and R. E. Tarjan, Computing an *st*-numbering, *Theoret. Comput. Sci.* **2** (1976), 339–344.
6. L. R. Ford and D. R. Fulkerson, "Flows in Networks," Princeton Univ. Press, Princeton, NJ, 1974.
7. T. Fujii, H. Horikawa, T. Kikuno, and N. Yoshida, A heuristic algorithm for gate assignment in one-dimensional array approach, *IEEE Trans. Computer-Aided Design* **6** (1987), 159–164.
8. M. R. Fellows and M. A. Langston, On well-partial-order theory and its application to combinatorial problems of VLSI design, *SIAM J. Discrete Math.* **5** (1992), 117–126.
9. M. R. Fellows and M. A. Langston, On search, decision and the efficiency of polynomial-time algorithms, *J. Comput. Syst. Sci.* **49** (1994), 769–779.
10. D. Fussell, V. Ramachandran, and R. Thurimella, Finding triconnected components by local replacements, in "Proceedings Sixteenth International Colloquium on Automata, Languages and Programming," vol. 372, 1989, pp. 379–393.
11. X. He, Efficient parallel algorithms for series parallel graphs, *J. Algorithms* **12** (1991), 409–430.
12. Y.-S. Hong, K.-H. Park, and M. Kim, Heuristic algorithms for ordering the columns in one-dimensional logic arrays, *IEEE Trans. Computer-Aided Design* **8** (1989), 547–562.
13. E. Korach and A. Tal, General vertex disjoint paths in series-parallel graphs, *Discrete Appl. Math.* **41** (1993), 147–164.

14. A. Lempel, S. Even, and I. Cederbaum, An algorithm for planarity testing of graphs, in "Theory of Graphs: International Symposium" (P. Rosenstiehl, Ed.), pp. 215-232, Gordon and Breach, New York, 1967.
15. P. C. Liu and R. C. Geldmacher, An  $O(\max(m, n))$  algorithm for finding a subgraph homeomorphic to  $K_4$ , *Congr. Numer.* **29** (1980), 597-609.
16. M. A. Langston and S. Ramachandramurthi, Dense layouts for series-parallel circuits, in "Proceedings, First Great Lakes Symposium on VLSI," 1991, pp. 14-17.
17. P. J. McGuinness and A. E. Kezdy, An algorithm to find a  $K_5$  minor, in "Proceedings, Third ACM-SIAM Symposium on Discrete Algorithms," 1992, pp. 345-356.
18. V. Ramachandran, private communication.
19. N. Robertson and P. D. Seymour, Graph minors XIII. The disjoint paths problem, *J. Combin. Theory Ser. B* **63** (1995), 65-110.
20. R. E. Tarjan, Depth-first search and linear graph algorithms, *SIAM J. Comput.* **1** (1972), 146-159.
21. J. Valdes, R. E. Tarjan, and E. Lawler, The recognition of series-parallel digraphs, *SIAM J. Comput.* **11** (1982), 298-313.

## On Search, Decision, and the Efficiency of Polynomial-Time Algorithms\*

MICHAEL R. FELLOWS

*Department of Computer Science, University of Victoria,  
Victoria, British Columbia, Canada V8W 2Y2*

AND

MICHAEL A. LANGSTON

*Department of Computer Science, University of Tennessee,  
Knoxville, Tennessee 37996-1301*

Received November 22, 1988; revised August 27, 1991

Recent advances in well-quasi-order theory have troubling consequences for those who would equate tractability with polynomial-time complexity. In particular, there is no guarantee that polynomial-time algorithms can be found just because a problem has been shown to be decidable in polynomial time. We present techniques for dealing with this unusual development. Our main results include a general construction strategy with which low-degree polynomial-time algorithms can now be produced for almost all of the catalogued algorithmic applications of well-quasi-order theory. We also prove that no such application of this theory can settle  $\mathcal{N} \stackrel{?}{=} \mathcal{NP}$  nonconstructively by any established method of argument.

© 1994 Academic Press, Inc.

### 1. INTRODUCTION

Although complexity theory is formulated in terms of decision problems, established techniques of algorithm design (with rare exception [Mi]) constructively address, instead, corresponding search or optimization versions of the problem at hand. In the vast majority of cases in which one knows that an algorithm exists to decide a problem in polynomial time, one knows precisely what the promised algorithm is. Furthermore, if the input is a "yes" instance, such an algorithm uncovers natural evidence (that is, an answer to the search version of the problem) as the basis for a positive decision.

\* A preliminary version of this paper was presented at the "21st ACM Symposium on Theory of Computing, Seattle, Washington, May 1989." This research is supported in part by the National Science Foundation under grant MIP-8919312, by the Office of Naval Research under contract N00014-90-J-1855, and by the Natural Sciences and Engineering Research Council of Canada under award 9820.

In contrast, advances in well-quasi-order theory, especially the seminal contributions of Robertson and Seymour, provide new and powerful nonconstructive tools for establishing polynomial-time decidability. These deep results suffer from some challenging difficulties:

- (1) the algorithms involve huge constants of proportionality,
- (2) the complexity of associated search problems is not established, and
- (3) there is no general means for finding (or even recognizing) correct algorithms.

We have developed a number of general techniques for dealing with each of these issues. At the editor's request, however, we suppress further discussion of issues (1) and (2) here. In the sequel, we concentrate only on issue (3).

Relevant background for this investigation is reviewed in the next section. Our main results are presented in Section 3, where we show how to devise constructive algorithms in the vast majority of applications. The result is that we can, in these cases, know a low-degree polynomial-time algorithm for search (and, hence, decision) without ever knowing the finite list of graphs on which the existence of the decision algorithm is based. Moreover, we show how to provide asymptotic optimality under very general circumstances. We also prove that, despite the nonconstructive nature of the underlying theory, this line of research cannot settle  $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$  nonconstructively by any established method of argument. A few concluding remarks make up the final section of this paper.

## 2. BACKGROUND

The graphs we consider are finite and undirected, but may have loops and multiple edges. A graph  $H$  is less than or equal to a graph  $G$  in the minor order, written  $H \leq_m G$ , if and only if a graph isomorphic to  $H$  can be obtained from  $G$  by a series of these two operations: taking a subgraph and contracting<sup>1</sup> an edge. A family  $F$  of graphs is said to be closed under the minor order if the facts that  $G$  is in  $F$  and that  $H \leq_m G$  together imply that  $H$  must be in  $F$ . The obstruction set for a family  $F$  of graphs is defined to be the set of graphs in the complement of  $F$  that are minimal in the minor order. If  $F$  is closed under the minor order, it has the following characterization:  $G$  is in  $F$  if and only if there exists no  $H$  in the obstruction set for  $F$  such that  $H \leq_m G$ .

A set along with a transitive, reflexive relation is called a quasi-order. For example, the class of all graphs under  $\leq_m$  is a quasi-order.<sup>2</sup> A quasi-ordered set  $(X, \leq)$  is well-quasi-ordered if (1) any subset of  $X$  has finitely many minimal

<sup>1</sup> An edge  $uv$  is contracted by deleting vertices  $u$  and  $v$  and adding a new vertex that is adjacent to each vertex that was originally adjacent either to  $u$  or  $v$ .

<sup>2</sup> Some authors have found it convenient to consider isomorphic graphs as distinct. Under this view, the minor order would not qualify as a partial order because it would not be anti-symmetric.

elements and (2)  $X$  contains no infinite descending chain  $x_1 > x_2 > x_3 > \dots$  of distinct elements.

**THEOREM 1 [RS4].** *Graphs are well-quasi-ordered under the minor relation.*

**THEOREM 2 [RS3].** *For every fixed graph  $H$ , the problem that takes as input a graph  $G$  and determines whether  $H \leq_m G$  is solvable in polynomial time.*

Theorem 1 is often called the Graph Minor Theorem. Theorem 2 ensures polynomial-time order tests. We term a well-quasi-ordered set with polynomial-time order tests a Robertson–Seymour set, or an RS set for short.

Theorems 1 and 2 guarantee only the existence of a polynomial-time decision algorithm for any minor-closed family of graphs. It has been shown that Theorem 1 is independent of constructive axiomatic systems and, indeed, any proof of it must use impredicative methods [FRS]. Also, there can be no systematic method of computing the finite obstruction set for an arbitrary minor-closed family  $F$  from the description of a Turing machine that accepts  $F$  (we prove this later).

A noteworthy feature of Theorem 2 is the low degree of the polynomials bounding the decision algorithms' running times. Letting  $n$  denote the number of vertices in  $G$ , the general bound is  $O(n^3)$ . If a minor-closed family excludes a planar graph, then it has bounded tree-width [RS1] and the bound is reduced to  $O(n \log n)$  [Re]. These polynomials possess enormous constants of proportionality, rendering them impractical for problems of any nontrivial size [RS2].

For an application of Theorems 1 and 2, consider the gate matrix layout problem [DKL]. Although the general problem is  $\mathcal{NP}$ -complete, it has been shown [FL1] that, for any fixed number of tracks, an arbitrary instance with  $n$  rows can be transformed into a graph such that the family of "yes" instances is closed under the minor order and excludes a planar graph. Thus the fixed-parameter version of gate matrix layout can be decided in  $O(n \log n)$  time.

A graph  $H$  is less than or equal to a graph  $G$  in the immersion order, written  $H \leq_i G$ , if and only if a graph isomorphic to  $H$  can be obtained from  $G$  by a series of these two operations: taking a subgraph and lifting<sup>3</sup> pairs of adjacent edges. The relation  $\leq_i$ , like  $\leq_m$ , defines a quasi-order on graphs with the associated notions of closure and obstruction sets.

**THEOREM 3 [RS1].** *Graphs are well-quasi-ordered under the immersion relation.*

**THEOREM 4 [FL3].** *For every fixed graph  $H$ , the problem that takes as input a graph  $G$  and determines whether  $H \leq_i G$  is solvable in polynomial time.*

Theorems 3 and 4, like Theorems 1 and 2, guarantee only the existence of a polynomial-time decision algorithm for any immersion-closed family of graphs. The method used to prove Theorem 4 yields a general time bound of  $O(n^{h+3})$ , where

<sup>3</sup> A pair of adjacent edges  $uv$  and  $vw$ , with  $u \neq v \neq w$ , is lifted by deleting the edges  $uv$  and  $vw$  and adding the edge  $uw$ .

$h$  denotes the order of the largest graph in the relevant obstruction set. With excluded-minor knowledge specific to an immersion-closed family, however, the time complexity for determining membership can in many cases be reduced to  $O(n \log n)$  by bounding the family's tree-width.

For an application of Theorems 3 and 4, consider the min cut linear arrangement problem [GJ]. Although the general problem is  $\mathcal{NP}$ -complete, it has been shown [FL3] that, for any fixed cutwidth, the family of "yes" instances is closed under the immersion order and has bounded tree-width. Thus the fixed-parameter version of min cut linear arrangement can be decided in  $O(N \log n)$  time.

### 3. CONSTRUCTIVIZATIONS

Decision algorithms based on finite obstruction sets do not decide by producing (or failing to produce) natural evidence and do not solve associated search problems. The situation may be modeled in terms of relations. Associated with a relation  $\Pi \subseteq \Sigma^* \times \Sigma^*$  are a number of basic computational problems that arise in the setting of well-quasi-ordered sets:

*checking*—the problem of determining, for input  $(x, y)$ , whether  $(x, y) \in \Pi$ ,

*decision*—the problem of determining, for input  $x$ , whether there exists a  $y$  such that  $(x, y) \in \Pi$ , and

*search*—the problem of computing a search function for  $\Pi$ , where such a function  $f: \Sigma^* \rightarrow \Sigma^* \cup \{\perp\}$  satisfies

- (1)  $f(x) = y$  implies that  $(x, y)$  is in  $\Pi$  and
- (2)  $f(x) = \perp \notin \Sigma$  implies that there exists no  $y$  for which  $(x, y)$  is in  $\Pi$ .

Search functions can often be computed by oracle algorithms that employ an algorithm for a related decision problem as the oracle.

**DEFINITION.** A self-reduction algorithm is an oracle algorithm that computes a search function for  $\Pi$  with oracle language domain  $(\Pi) = \{x \mid \text{there exists a } y \text{ for which } (x, y) \text{ is in } \Pi\}$ . The overhead of such an algorithm is its time complexity as measured by charging each oracle invocation with only a unit-time cost.

**DEFINITION.** A quasi-order  $(R, \leq)$  is uniformly enumerable if there is a recursive enumeration  $(r_0, r_1, r_2, \dots)$  of  $R$  with the property that  $r_i \leq r_j$  implies  $i \leq j$ .

In the minor and immersion orders, for example, a natural uniform enumeration is to generate all finite graphs based on a monotonically nondecreasing sequence of number of vertices, with graphs having the same number of vertices generated based on a monotonically nondecreasing sequence of number of edges, with ties

(graphs with the same number of vertices and the same number of edges) broken arbitrarily.

**DEFINITION.** Under a uniform enumeration of the elements of domain ( $\Pi$ ), we say that a self-reduction algorithm for  $\Pi$  is *uniform* if, on input  $r_j$ , the oracle for domain ( $\Pi$ ) is consulted concerning  $r_i$  only for  $i \leq j$ .

**DEFINITION.** An oracle algorithm is *honest* if, on inputs of size  $n$ , its oracle is consulted concerning only instances of size  $O(n)$ .

**DEFINITION.** An oracle algorithm  $A$  with overhead bounded by  $T(n)$  is *robust* (with respect to  $T(n)$ ) if  $A$  is guaranteed to halt within  $T(n)$  steps for any oracle language.

Despite the nonconstructivity inherent in the tools discussed in the previous section, we now show that low-degree polynomial-time search (and hence decision) algorithms can often be constructed. The general technique we present works in a rather surprising fashion: we are able to write down a correct algorithm without knowing the complete relevant obstruction set and, in some cases, without knowing the exact polynomial that bounds the running time of the algorithm.

**THEOREM 5.** *Let  $F = \text{domain}(\Pi)$  be a closed family in a uniformly enumerable well-quasi-order, and suppose the following are known:*

- (1) *an algorithm that solves the checking problem for  $\Pi$  in  $O(T_1(n))$  time,*
- (2) *order tests that require  $O(T_2(n))$  time,*
- (3) *a uniform self-reduction algorithm (its time bound is immaterial), and*
- (4) *an honest robust self-reduction algorithm that requires  $O(T_3(n))$  overhead.*

*Then an algorithm requiring  $O(\max\{T_1(n), T_2(n) \cdot T_3(n)\})$  time is known that solves the search problem for  $\Pi$ .*

*Proof.* Let  $I$  denote an arbitrary input instance and let  $K$  denote the known elements of the (finite) obstruction set for  $F$ . (Initially,  $K$  can be empty.) We treat  $K$  as if it were the correct obstruction set, that is, as if we did know a correct decision algorithm. Since the elements of  $K$  will always be obstructions, if we find that  $H \leq I$  for some  $H \in K$ , then our algorithm reports "no" and halts. Otherwise, after checking each element of  $K$  to confirm that it is not less than or equal to  $I$ , we attempt to self-reduce and check the solution so obtained. If the solution is correct, then our algorithm reports "yes," outputs the solution and halts.

In general, however, it may turn out that our check of the solution reveals that we have self-reduced to a nonsolution. This can only mean that there is at least one obstruction  $H \notin K$ . In this event, we proceed by generating the elements of  $R$  in uniform order until we find a new obstruction (an element that properly contains no other obstruction but that cannot be uniformly self-reduced to a solution).

When such an obstruction is encountered, we need only augment  $K$  with it and start over on  $I$ .

Let  $C_1$  denote the cardinality of the correct obstruction set and let  $C_2$  denote the largest number of vertices in any of its elements. Then, for some suitably chosen function  $f$ , the total time spent by this algorithm is bounded by  $O(C_1(T_1(n) + T_2(n) \cdot T_2(n)) + f(C_2))$ . ■

Theorem 5 prompts several observations.

*Observation 1.* Most (but, interestingly, not all) of the known RS set applications that ensure polynomial-time problem complexity can be made constructive. This follows because polynomial-time order tests are known for the minor and immersion orders and because we know, in most cases, low-degree polynomial-time algorithms for checking and for (uniform and honest robust) self-reducing. Reconsider, for example, the min cut linear arrangement problem. Checking a candidate solution is easily performed in  $O(n)$  time. Order tests are  $O(n \log n)$ . Uniform self-reduction is achievable (although it is rather cumbersome). Honest robust self-reduction can be performed with  $O(n)$  overhead. Therefore, Theorem 5 provides the following constructive corollary: for any fixed  $k$ , the search and decision versions of min cut linear arrangement can be solved in  $O(n^2 \log n)$  time with a known algorithm.

*Observation 2.* For problems such as knotlessness [FL2], for which no algorithm (with any time bound) for the decision problem is constructively known, but for which (only super-exponential) algorithms for the checking problem are constructively known [Ha], we have an unexpected situation. Finding a uniform self-reduction, a task seemingly very different from decision, would provide the first known decision algorithm for this problem.

*Observation 3.* In Theorem 5, it is of course possible to replace the hypothesis that a uniform self-reduction is known with the alternate hypothesis that a decision algorithm is known. (In fact, due to Theorem 5 itself, it follows that this new hypothesis is potentially weaker.) In some applications, this may be more convenient.

*Observation 4.* Although an attempt to implement the algorithm used in the proof of Theorem 5 appears at first to be out of the question, closer inspection reveals that it may in fact provide the basis for viable and wholly novel approaches for solving a number of practical problems. This scheme can be viewed as a learning algorithm that gradually accumulates a useful subset of the obstruction set, and invokes an exhaustive learning component only when forced to do so by input that it cannot handle with this subset. Furthermore, some obstructions are already known or are easy to identify for most problems, so that we need not start with the empty set. Growing evidence appears to indicate that, for many problems amenable to RS set theory, a relatively small collection of obstructions is often enough to

support search and decision algorithms on input generally encountered in practice (see, for example, [LR]).

Our next result extends an entertaining (but completely unimplementable) idea first observed in [Le] and explicitly proved in [Sc] to the setting of well-quasi-ordered sets. The original idea applies only to the computation of search functions restricted to  $\text{domain}(\Pi)$ . Where  $\text{domain}(\Pi)$  is closed in a well-quasi-order, this restriction can be lifted.

**THEOREM 6.** *Let  $F = \text{domain}(\Pi)$  be a closed family in a uniformly enumerable well-quasi-order, and suppose the following are known:*

- (1) *an algorithm that solves the checking problem for  $\Pi$  in  $O(T_1(n))$  time,*
- (2) *order tests that require  $O(T_2(n))$  time, and*
- (3) *a uniform self-reduction algorithm (its time bound is immaterial).*

*Then an algorithm requiring  $O(\max\{T_0(n) + T_1(n) \cdot \log T_0(n), T_2(n)\})$  time is known that solves the search problem for  $\Pi$ , where  $T_0(n)$  denotes the time complexity of any algorithm solving this search problem.*

*Proof.* Condition (3) ensures that at least one search algorithm exists, so that  $T_0$  is defined. We interleave the following two operations. In case the input is in  $F$ , we employ the exponential form of diagonalization from [Le],<sup>4</sup> which requires time proportional to  $2^X T_0(n) + (X + \log T_0(n)) T_1(n)$ , where  $X$  denotes the index of the lowest-indexed Turing machine solving the search problem in time  $T_0(n)$ . In case the input is in  $\bar{F}$ , we employ the uniform enumeration of the elements of the order (along with obstruction containment tests) as we did in the proof of Theorem 5. ■

Thus one theoretically attains asymptotic optimality, albeit at the cost of explosive constants of proportionality. A curious feature of the preceding theorem is that provision is made neither for computing the relevant obstruction set nor for determining the function  $T_0$ . That is, even if one could implement the algorithm, no bounded amount of computation would necessarily reveal when one has encountered the last obstruction or when an optimal algorithm begins to outperform all others.

To preface the final result of this section, we observe that many members of the research community have invested considerable effort in producing  $\mathcal{NP}$ -completeness proofs for a vast array of seemingly difficult combinatorial problems. Such proofs implicitly rely on the assumption that, if  $\mathcal{P} = \mathcal{NP}$ , an  $\mathcal{NP}$ -completeness proof is not in vain, but instead becomes a polynomial-time algorithm. There has always been a potential flaw in this logic, namely that a proof of  $\mathcal{P} = \mathcal{NP}$  might be nonconstructive. For example, what if chromatic number with, say, fixed  $k = 3$  were minor-closed (which is it not)? After all, traditional methods for attacking the

<sup>4</sup> Turing machines are emulated in phases. During phase  $i$ , each machine whose index  $h$  lies in the range  $[1, i]$  is emulated until it has performed  $2^{i-h}$  steps.

$\mathcal{P} = \mathcal{NP}$  question have to date failed; one might expect that if the issue is ever to be resolved, new techniques must be brought to bear. We now show that such a vexing outcome cannot happen based on well-quasi-ordered sets and known problem reduction schemes. Recall that an RS set is a well-quasi-ordered set that supports polynomial-time order sets. We make the additional assumption that these order tests are known (as they are in the minor and immersion orders).

**DEFINITION.** By the statement  $\mathcal{P}$  is constructively equal to  $\mathcal{NP}$  we mean that an algorithm is known that computes, from the index and time bound<sup>5</sup> of a nondeterministic polynomial-time Turing machine that recognizes a set  $X$ , the index of a deterministic polynomial-time Turing machine that recognizes  $X$ . A set  $X$  is constructively  $\mathcal{NP}$ -hard if a polynomial-time many-to-one reduction from satisfiability (SAT) to  $X$  is known.

Every problem presently known to be  $\mathcal{NP}$ -hard is constructively  $\mathcal{NP}$ -hard as well, simply because the relevant reductions are constructively known (rather than only known to exist). We now demonstrate that it is not possible to prove  $\mathcal{P} = \mathcal{NP}$  by an obvious approach, such as searching for a known  $\mathcal{NP}$ -complete graph problem that can be shown to be minor-closed, except in a constructive way.

**THEOREM 7.** *Let  $F$  denote a closed family in a uniformly enumerable RS set. If it is constructively  $\mathcal{NP}$ -hard to determine membership in  $F$ , then  $\mathcal{P}$  is constructively equal to  $\mathcal{NP}$ .*

*Proof.* Let  $i$  denote the index of a nondeterministic Turing machine that accepts language  $L$  in time bounded by polynomial  $p$ . We compute the index  $i'$  of a deterministic polynomial-time Turing machine that accepts  $L$  by describing a polynomial-time algorithm for recognizing  $L$ . For input  $x$ , we can of course use  $i$  and  $p$  to compute (by a known algorithm) in time polynomial in  $|x|$  a Boolean expression  $E_x$  that is satisfiable if and only if  $x \in L$ . It is enough to argue that knowing a reduction  $f$  from SAT to  $F$ -membership yields a known polynomial-time algorithm for SAT.

Note that an honest robust (and uniform) polynomial-time self-reduction algorithm for SAT is easily described, by simply taking the usual self-reduction and guarding against a faulty oracle, precomputing the number of oracle calls required when the oracle is trustworthy.

Let  $\text{Im}^+$  denote the set of all graphs that are images (under the many-to-one reduction  $f$ ) of satisfiable expressions, and let  $\text{Im}^-$  denote the set of graphs that are images of unsatisfiable expressions. Thus,  $\text{Im}^+ \subseteq F$  and  $\text{Im}^- \subseteq \bar{F}$ . Since  $F$  is closed,  $\text{Im}^+$  is closed in  $\text{Im} = \text{Im}^+ \cup \text{Im}^-$ , under the inherited order. Let  $O_{\text{SAT}}$  denote the minimal elements of  $\text{Im}^-$ . The set  $O_{\text{SAT}}$  is finite, by the well-quasi-ordering of an RS set.

<sup>5</sup> Observe that knowledge of a polynomial bound on acceptance time is necessary; even the proof of Cook's Theorem is nonconstructive without a known bound.

Let  $E$  denote an arbitrary Boolean expression. Clearly,  $E$  is satisfiable if and only if  $f(E) \geq y$  for each  $y \in O_{\text{SAT}}$ . Let  $E_1, E_2, \dots$  be a recursive enumeration of all Boolean expressions, and let  $O$  denote the known candidates for  $O_{\text{SAT}}$ . (Initially,  $O$  can be empty.) Each element of  $O$  will be greater than or equal to some obstruction, so we treat  $O$  as if it were  $O_{\text{SAT}}$ . We begin by generating expressions in the enumeration and exhaustively determining whether each is satisfiable until we encounter an  $E_j$  that is unsatisfiable. Resetting  $O$  to be the minimal elements of  $O \cup \{f(E_j)\}$ , we use the known order tests to learn whether  $f(E)$  contains an element of  $O$ . If it does, then our algorithm reports "no" and halts. Otherwise, we attempt to self-reduce  $E$  using  $f$ , together with the order tests, to implement the oracle. If we succeed in producing a satisfying truth assignment, then our algorithm reports "yes" and halts. If the question remains unsettled (that is, the attempted self-reduction has failed, but there is no  $y \in O$  for which  $f(E) \geq y$ ), then we resume generating expressions until we can augment  $O$  and start over on  $E$ .

Since  $O_{\text{SAT}}$  is finite, we are guaranteed to achieve  $O = O_{\text{SAT}}$  within some bounded initial segment of the enumeration, although evidence for a correct decision concerning  $E$  may be produced well before that point is reached. The running time of the algorithm is bounded by a polynomial function of  $|x|$ . ■

The method used to prove Theorem 7 can be viewed as an extension of the technique employed in the proof of Theorem 5. It has recently been suggested that there may be an alternate proof based on properties of sparse sets [CG].

#### 4. CONCLUSIONS

Our construction techniques do not depend on knowing, nor do they provide a means for computing, the relevant (finite) obstruction sets. An obvious question arises: can these sets be systematically computed? The following theorem shows that, in a general sense, they cannot. (We state this result for the minor order of finite graphs; the proof can be easily modified to handle other RS sets.)

**THEOREM 8.** *There is no algorithm to compute, from a finite description of a minor-closed family  $F$  of graphs as represented by a Turing machine that accepts precisely the graphs in  $F$ , the set of obstructions for  $F$ .*

*Proof.* We reduce from the Halting Problem. Given a Turing machine  $M$  and a word  $x$ , we can determine whether  $M$  halts on  $x$  as follows. We modify the description of  $M$  to obtain a description of a Turing machine  $M'$  that embodies the following algorithm. Given as input a graph  $G$ ,  $M'$  first computes the index  $i(G)$  of  $G$  in a recursive enumeration of all finite graphs that is uniform with respect to the minor order.  $M'$  then simulates  $M$  on input  $x$  for  $i(G)$  steps.

If  $M$  does not halt in at most  $i(G)$  steps, then  $M'$  accepts  $G$ . Otherwise, if  $M$  halts in exactly  $i(G)$  steps, then  $M'$  rejects  $G$ . If the halting time of  $M$  on input  $x$  is

$t < i(G)$ , then  $M'$  determines the graph  $H$  such that  $i(H) = t$ , and tests whether  $G \geq_m H$ . If  $G \geq_m H$ , then  $M'$  rejects  $G$ . Otherwise,  $M'$  accepts  $G$ .

We claim that  $M'$  accepts a minor-closed family of graphs. If  $M$  does not halt on input  $x$ , then  $M'$  accepts all graphs, which is trivially minor-closed. If the halting time of  $M$  on input  $x$  is  $t$ , then  $M'$  accepts precisely those graphs  $G$  for which  $G \not\geq_m H$ , a minor-closed family with the single obstruction  $H$ . To see this, suppose  $G \geq_m H$ . Since the enumeration is uniform,  $i(G) \geq t = i(H)$ , and so  $G$  is rejected by  $M'$ . If  $G \not\geq_m H$ , then  $G$  is accepted by  $M'$ , regardless.

The description of  $M'$  is clearly computable from  $x$  and the description of  $M$ . Whether  $M$  halts on  $x$  can thus be determined by computing the obstruction set of the family of graphs accepted by  $M'$ , since this obstruction set is empty if and only if  $M$  fails to halt on input  $x$ . ■

#### ACKNOWLEDGMENTS

We thank Vijaya Ramachandran for bringing to our attention Leonid Levin's diagonalization idea from [Le]. We also acknowledge those who have participated in technical discussions on this general subject, including Dan Archdeacon, Mike Attalah, Manuel Blum, Donna Brown, Randy Bryant, Amos Fiat, Victor Klee, Gene Lawler, Dave Liu, John Reif, Neil Robertson, Arny Rosenberg, and Paul Seymour.

#### REFERENCES

- [CG] S. COOK AND A. GUPTA, private communication.
- [DKL] N. DEO, M.S. KRISHNAMOORTHY, AND M.A. LANGSTON, Exact and approximate solutions for the gate matrix layout problem, *IEEE Trans. Computer-Aided Design* **6** (1987), 79–84.
- [FL1] M. R. FELLOWS AND M. A. LANGSTON, Nonconstructive advances in polynomial-time complexity, *Inform. Proc. Lett.* **26** (1987), 157–162.
- [FL2] M. R. FELLOWS AND M. A. LANGSTON, Nonconstructive tools for proving polynomial-time decidability, *J. Assoc. Comput. Mach.* **35** (1988), 727–739.
- [FL3] M. R. FELLOWS AND M. A. LANGSTON, On well-partitional-order theory and its application to combinatorial problems of VLSI design, *SIAM J. Discrete Math.* **5** (1992), 117–126.
- [FRS] H. FRIEDMAN, N. ROBERTSON AND P. D. SEYMOUR, The metamathematics of the graph minor theorem, *Contemporary Math.* **65** (1987), 229–261.
- [GJ] M. R. GAREY AND D. S. JOHNSON, “*Computers and Intractability*,” Freeman, San Francisco, 1979.
- [Ha] W. HAKEN, Theory der Normalflächen, *Acta Math.* **105** (1961), 245–375.
- [Le] L. A. LEVIN, Universal enumeration problems, *Problemy Peredachi Informatsii* **2** (1972), 115–116. [in Russian]
- [LR] M. A. LANGSTON AND S. RAMACHANDRAMURTHI, Dense layouts for series-parallel circuits, in “Proceedings, 1991 Great Lake Symposium on VLSI”, pp. 14–17.
- [Mi] G. L. MILLER, Rieman's hypothesis and tests for primality, *J. Comput. System Sci.* **13** (1976), 300–317.
- [Re] B. REED, Finding approximate separators and computing with tree width quickly, in “Proceedings, 1992 Symposium on Theory of Computing,” pp. 221–228.
- [RS1] N. ROBERTSON AND P. D. SEYMOUR, Graph minors. IV. Tree-width and well-quasi-ordering, *J. Combin. Theory Ser. B* **48** (1990), 227–254.

- [RS2] N. ROBERTSON AND P. D. SEYMOUR, Graph minors. V. Excluding a planar graph, *J. Combin. Theory Ser. B.* **41** (1986), 92–114.
- [RS3] N. ROBERTSON AND P. D. SEYMOUR, Graph minors. XIII. The disjoint paths problem, to appear.
- [RS4] N. ROBERTSON AND P. D. SEYMOUR, Graph minors. XVI. Wagners conjecture, to appear.
- [Sc] C. P. SCHNORR, Optimal algorithms for self-reducible problems in "Proceedings, 1976 International Conference on Automata, Programming and Languages," pp. 322–337.

## ANALYSIS OF A COMPOUND BIN PACKING ALGORITHM\*

DONALD K. FRIESEN† AND MICHAEL A. LANGSTON‡

**Abstract.** Consider the classic bin packing problem, in which we seek to pack a list of items into the minimum number of unit-capacity bins. The worst-case performance of a *compound* bin packing algorithm that selects the better packing produced by two previously analyzed heuristics, namely, FFD (first fit decreasing) and B2F (best two fit) is investigated. FFD and B2F can asymptotically require as many as  $\frac{11}{9}$  and  $\frac{5}{4}$  times the optimal number of bins, respectively. A new technique, *weighting function averaging*, is introduced to prove that our compound algorithm is superior to the individual heuristics on which it is based, never using more than  $\frac{6}{5}$  times the optimal number of bins.

**Key words.** bin packing, compound algorithms, heuristics, weighting functions, worst-case analysis

**AMS(MOS) subject classifications.** 68Q20, 68Q25

**1. Introduction.** In the usual definition of the bin packing problem, we seek to pack the items of a list  $L = \{l_1, l_2, \dots, l_N\}$ , each item with size in the range  $(0, 1]$ , into the minimum number of unit-capacity bins. It is easily verified that this problem is NP-hard. Therefore, we focus our efforts on practical, efficient approximation algorithms in hopes of guaranteeing near-optimal results. (Note that there are algorithms guaranteed to produce results as close to the optimum as desired [1], [7]. Unfortunately, these algorithms are not practical to implement because the time required to ensure results at most  $(1 + \varepsilon)$  times the optimum grows extremely rapidly as  $\varepsilon$  approaches zero.)

We use worst-case analysis as a measure of the worth of a bin packing heuristic. The heuristic may not discover the best packing, but we endeavor to show that it always provides results close to the optimum. For some algorithm, ALG, let  $\text{ALG}(L)$  represent the number of nonempty bins required by ALG to pack  $L$ . For instance,  $\text{OPT}(L)$  denotes the number of bins required in an optimal packing of  $L$ . We restrict our attention to two off-line<sup>1</sup> algorithms: FFD (first fit decreasing) and B2F (best two fit). Given any list  $L$ , it is known from [6] that  $\text{FFD}(L)$  does not exceed  $(\frac{11}{9}) \text{OPT}(L) + 4$ , and from the Appendix to this paper that  $\text{B2F}(L)$  does not exceed  $(\frac{5}{4}) \text{OPT}(L) + 4$ . Moreover, examples exist that demonstrate that these bounds are asymptotically tight.

It seems reasonable to suggest that these two heuristics produce particularly inferior packings for rather small, distinct regions of the input space. Based on this conjecture, we analyze a *compound* algorithm, CFB, in which both FFD and B2F are applied and the better packing selected. This notion of combining two or more heuristics is an attractive one, but the analysis of such an algorithm can be especially difficult; only a few compound algorithms have been successfully analyzed in the literature (see, for example, [2], [8], [9]). We note that a tight worst-case bound of  $71/60$  has recently been reported for a modification of the FFD algorithm [5], thereby yielding the lowest bound yet published for an efficient bin packing heuristic. This bound is superior to the upper bound of  $\frac{6}{5}$  that we prove here, but is inferior to the lower bound of  $227/195$  provided by the worst

\* Received by the editors June 19, 1989; accepted for publication (in revised form) May 8, 1990. A preliminary version of a portion of this paper was presented at the twentieth Allerton Conference on Communication, Control, and Computing held in Monticello, Illinois in October, 1982.

† Department of Computer Science, Texas A&M University, College Station, Texas 77843.

‡ Department of Computer Science, University of Tennessee, Knoxville, Tennessee 37996-1301 and Department of Computer Science, Washington State University, Pullman, Washington 99164-1210. This research was supported in part by National Science Foundation grants MIP-8603879 and MIP-8919312, and by Office of Naval Research contract N00014-88-K-0343.

<sup>1</sup> An off-line algorithm is free to preview and rearrange items before it begins to pack them.

examples we know of for CFB. Moreover, the novel analysis we devise for our compound algorithm merits attention and may, we hope, be applicable in other settings.

We shall employ the technique of “weighting”  $L$  so that the FFD and B2F packings can be compared to an optimal packing. Although we would like to determine the minimum of  $\{\text{FFD}(L), \text{B2F}(L)\}$ , the analysis involved is extremely complicated. Instead, we investigate the average of  $\{\text{FFD}(L), \text{B2F}(L)\}$ , in an effort to obtain a weak upper bound on the minimum. In particular we show that, after eliminating certain cases where we can guarantee that one or the other algorithm performs within our bound of  $\frac{6}{5}$ , our weighting of  $L$  ensures that the average and hence the minimum number of bins used by the two algorithms is within the bound.

In the next section, we present some preliminary analysis and demonstrate that  $\text{CFB}(L)$  can be as great as  $(227/195) \text{OPT}(L)$ . We also introduce a typing scheme for the items of  $L$  based on size. In § 3, we establish the specific conditions required for the FFD packing to use more than  $\frac{6}{5}$  the optimal number of bins. Section 4 contains an analogous determination for B2F. We present our main result in § 5, proving that  $\text{CFB}(L)$  does not exceed  $(\frac{6}{5}) \text{OPT}(L) + 8$ . The final section contains remarks about proving a tighter performance bound for CFB. In the Appendix, we discuss in further detail the B2F algorithm and derive its asymptotic worst-case bound.

**2. Preliminary discussion.** We begin by describing the FFD and B2F heuristics more precisely. The FFD algorithm can be implemented by first sorting all items so that their sizes are arranged in nonincreasing order. Each bin is packed by repeatedly placing in it the largest unpacked item that fits. When no more items are available that fit, the next bin is packed. The B2F algorithm modifies this in the following way. First a bin is packed as by the FFD rule. If the bin contains more than a single item, then the list is checked to see if the smallest item in the bin could be replaced by two items that would pack the bin more nearly full. If so, those two whose sum is largest are used in place of the smallest item in the bin. A number of other schemes could be used to decide which two replace the smallest item, but almost any choice will satisfy our analysis, subject to the following modification made to simplify the proof: items of sizes less than or equal to  $\frac{1}{6}$  will be held back until all larger items are packed. An FFD-like procedure is used to complete the packing when only items of size no greater than  $\frac{1}{6}$  are left. The purpose of this modification is to reduce the number of combinations to consider in proving an asymptotic  $\frac{6}{5}$  bound, although it seems likely that this modification actually detracts somewhat from the performance of the compound algorithm.

Figure 1 depicts the worst example (independent, of course, of an additive constant) that we were able to contrive for the CFB algorithm. For simplicity, the bin size has been expanded to 559. All of the examples we devised that were even close to being this poor were dependent on the small items being held back, so that the FFD and B2F packings are the same.

We denote the size of an item  $l_i \in L$  by  $s(l_i)$ . Thus, after sorting,  $s(l_1) \geq s(l_2) \geq \dots \geq s(l_N)$ . We use  $last$  to denote the index of the last item packed by FFD. Note that  $l_{last}$  may not be the smallest item in  $L$ , since smaller items may have been packed earlier where  $l_{last}$  did not fit.

To prove that  $\frac{6}{5}$  is an asymptotic upper bound on the worst-case behavior of CFB, we now proceed by contradiction and henceforth assume that  $L$  denotes a counterexample. That is, we assume that both  $\text{FFD}(L)$  and  $\text{B2F}(L)$  exceed  $(\frac{6}{5}) \text{OPT}(L) + 8$ . Without loss of generality, we also assume that  $L$  is minimal. By this we mean that no counterexample exists with which  $\text{OPT}$  can use fewer bins, and that no counterexample is possible with fewer items for this minimal number of bins. (Of course, minimality for CFB does not imply minimality for either FFD or B2F alone.)

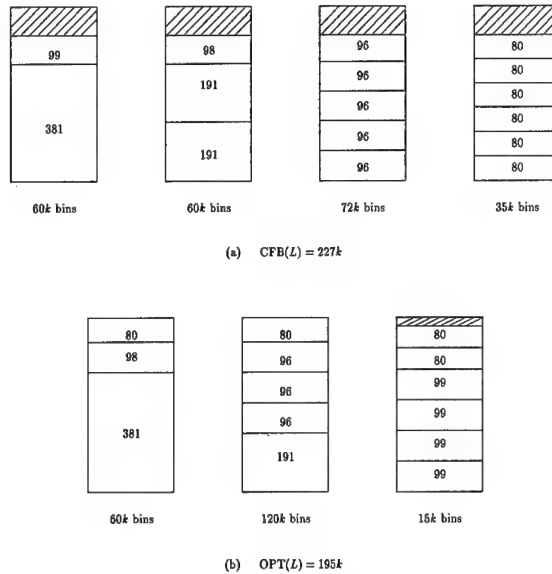


FIG. 1. Example for which  $\text{CFB}(L) = (227/195) \text{OPT}(L)$ , using bin size 559.

An immediate consequence of this is that  $L$  contains no item whose size is less than or equal to  $\frac{1}{6}$ . If it did, then minimality requires that one or more such items must be packed in the last bin by either the FFD or the B2F algorithm, in which case all preceding bins would be packed to a level of at least  $\frac{5}{6}$ . A simple "conservation of size" argument ensures that, for such a list, no packing could use fewer than  $(\frac{5}{6})(\text{CFB}(L) - 1)$  bins.

With this in mind, we let  $s(l_{\text{last}}) = \frac{1}{6} + \Delta$ , for some  $\Delta > 0$ . Since no item has size less than or equal to  $\frac{1}{6}$ , we know that no bin in any packing of  $L$  has more than five items.

We use the notation  $B^*$  for an arbitrary bin of the optimal packing, and  $|B^*|$  to denote the number of items  $B^*$  contains. For the bins of the FFD or B2F packing, we use  $B_1, B_2, \dots$  as the sequence of bins in the order in which they are packed.

LEMMA 2.1.  $L$  contains no item  $l_i$  with  $s(l_i) \geq \frac{2}{3}$ .

*Proof.* To obtain the proof, assume otherwise. In both the FFD and B2F packings, the largest item  $l_1$  is packed in  $B_1$  with at most one other item, the largest that would fit. The optimal bin containing  $l_1$  can contain at most one additional item and in fact can be packed no better than  $B_1$ . If the item or items of  $B_1$  are removed from  $L$ , then all three of  $\text{FFD}(L)$ ,  $\text{B2F}(L)$ , and  $\text{OPT}(L)$  can be reduced by one, contradicting the presumed minimality of  $L$  with respect to CFB.  $\square$

There can be no bin containing only one item in the FFD packing (except, possibly, for the last bin). If there were,  $s(l_{\text{last}})$  must exceed  $\frac{1}{3}$ , since otherwise  $l_{\text{last}}$  would have fit, and it is known that  $\text{FFD}(L)$  is bounded by  $(\frac{7}{6}) \text{OPT}(L) + 2$  whenever  $s(l_{\text{last}})$  exceeds  $\frac{1}{4}$ . (See [6, Thm. 4.10].) From this it also follows that  $\Delta$  must be less than or equal to  $\frac{1}{12}$ .

Each item of  $L$  is assigned a type as shown in Table 1. Although this typing scheme is motivated by the structure of a typical packing produced by the FFD rule (more will be said on this in the next section), we classify items exclusively by their size so that we can compare both  $\text{FFD}(L)$  and  $\text{B2F}(L)$  to  $\text{OPT}(L)$ . Note that  $\Delta$  cannot exceed  $\frac{1}{30}$  if  $Y_4$  or  $X_5$  items exist.

**3. A close look at FFD.** We say that an item is "regular" if there is no larger item available when it is packed. A "fallback" item is one that is packed when one or more

TABLE 1  
Item types based on size.

Type	Min size	Max size
$Y_1$	$> \frac{1}{2}$	$< \frac{2}{3}$
$X_2$	$> \frac{5}{12} - \Delta/2$	$\leq \frac{1}{2}$
$Y_2$	$> \frac{1}{3}$	$\leq \frac{5}{12} - \Delta/2$
$X_3$	$> \frac{5}{18} - \Delta/3$	$\leq \frac{1}{3}$
$Y_3$	$> \frac{1}{4}$	$\leq \frac{5}{18} - \Delta/3$
$X_4$	$> \frac{5}{24} - \Delta/4$	$\leq \frac{1}{4}$
$Y_4$	$> \frac{1}{5}$	$\leq \frac{5}{24} - \Delta/4$
$X_5$	$> \frac{1}{6}$	$\leq \frac{1}{5}$

larger items are available. Thus the notation we have used in Table 1 roughly agrees with the way items are packed by FFD. That is, regular items of type  $X_i$  are generally packed by FFD in a bin consisting of the  $i$  largest items available when the bin is packed. We call such a bin an  $X_i$  bin. Regular items of type  $Y_i$  are generally packed with  $i - 1$  other  $Y_i$  items and a (smaller) fallback item. We call such a bin a  $Y_i$  bin. (Note that no  $Y_i$  bin,  $i \geq 2$ , can have more than one fallback item, as the following argument shows. If two fallback items are used, then they combine to fill more than  $\frac{1}{3}$  of the bin. In this event, however, the two or more regular items fill less than  $\frac{2}{3}$  of the bin, and the smaller regular item has a size less than  $\frac{1}{3}$ , implying that another regular item would have fit in the bin as well.)

This motivates the range of sizes we have selected for each item type. For example, the sum of the sizes of the two items in an  $X_2$  bin must exceed  $1 - (\frac{1}{6} + \Delta)$ , or else  $l_{\text{last}}$  would have been used as a fallback item in that bin. Hence, with the exception of items from the first or last  $X_2$  bin, every regular  $X_2$  item must have a size in the range  $(\frac{5}{12} - \Delta/2, \frac{1}{2}]$ . Similar size restrictions are used to define the other item types as summarized in Table 1. We use these same size ranges to assign a type to each fallback item.

There may also be some bins, which we define as “exceptional” for the FFD packing, that are not packed by FFD with items of the expected sizes. These can only be the first or last bins of a particular type, subject to the following constraints. If the last bin of type  $Y_i$  is exceptional (that is, it does not contain  $i$  items of type  $Y_i$ ), then the next bin is an  $X_{i+1}$  bin that is not exceptional if there are at least two  $X_{i+1}$  bins. Similarly, if the last bin of type  $X_i$  is exceptional, then the first bin of type  $Y_i$  is not exceptional unless it is also the last  $Y_i$  bin.

Consequently, there are at most eight exceptional bins in the FFD packing, including the last bin packed (which contains  $l_{\text{last}}$ ). We define an exceptional item to be one packed in an exceptional bin or one smaller than  $l_{\text{last}}$ .

We now seek to determine the precise conditions necessary for FFD( $L$ ) to exceed  $(\frac{6}{5})\text{OPT}(L) + 8$ . In this effort, we employ a weighting function  $w_F: L \rightarrow \mathbf{R}^+$ . We extend  $w$  to subsets of  $L$  in the obvious fashion. For example,  $w_F(B_j)$  denotes  $\sum_{l_i \in B_j} w_F(l_i)$ . Our intent is to assign each item as small a weight as possible and yet ensure that the weight of any nonexceptional FFD packed bin is at least 1. Table 2 describes our definition of  $w_F$  for nonexceptional items.

Recall that fallback items, like regular items, are assigned a type based on their size. We deviate slightly from this definition of  $w_F$  for items packed in  $Y_1$  bins. Consider any two  $Y_1$  items  $a$  and  $b$ , where  $a$  precedes  $b$ . Since  $s(a) \geq s(b)$ , we increase  $w(a)$ , if necessary, to ensure that  $w(a) \geq w(b)$  and reduce the weight of any item(s) packed with  $a$  accordingly. For future reference, we state this formally as follows:

TABLE 2  
Weighting function  $w_F$  based on FFD packing.

Type of nonexceptional items in an FFD-packed bin	Weights assigned
$Y_1$ , any two items	$\frac{3}{5}, \frac{1}{5}, \frac{1}{5}$
$Y_1, X_2$	$\frac{3}{5}, \frac{2}{5}$
$Y_1, Y_2$	$\frac{3}{5}, \frac{2}{5}$ if $\exists Y_2$ bin(s), else $\frac{2}{3}, \frac{1}{3}$ if $s(Y_1) \leq \frac{2}{3} - 2\Delta$ , else $\frac{11}{15}, \frac{4}{15}$
$Y_1, X_3$	$\frac{2}{3}, \frac{1}{3}$
$Y_1, Y_3$	$\frac{11}{15}, \frac{4}{15}$
$Y_1, X_4$	$\frac{3}{4}, \frac{1}{4}$
$Y_1, Y_4$ or smaller item	$\frac{4}{5}, \frac{1}{5}$
$X_2, X_2$	$\frac{1}{2}, \frac{1}{2}$
$Y_2, Y_2, X_3$	$\frac{11}{30}, \frac{11}{30}, \frac{4}{15}$
$Y_2, Y_2, Y_3$ or smaller item	$\frac{2}{5}, \frac{2}{5}, \frac{1}{5}$
$X_3, X_3, X_3$	$\frac{1}{3}, \frac{1}{3}, \frac{1}{3}$
$Y_3, Y_3, Y_3$ , any item	$\frac{4}{15}, \frac{4}{15}, \frac{4}{15}, \frac{1}{5}$
$X_4, X_4, X_4, X_4$	$\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}$
any five items	$\frac{1}{5}, \frac{1}{5}, \frac{1}{5}, \frac{1}{5}, \frac{1}{5}$

$Y_1$  weighting rule: If  $a$  and  $b$  are  $Y_1$  items, and  $a$  is packed in a bin before the bin containing  $b$ , then  $w_F(a) \geq w_F(b)$ .

An exceptional item receives a weight of zero, completing our definition of  $w_F$ . For the convenience of the reader, Table 3 provides a listing of the possible weights for each nonexceptional item type.

LEMMA 3.1. *The FFD weight of an optimal bin cannot exceed  $\frac{6}{5}$  unless the bin contains a  $Y_1$  item or a  $Y_2$  item whose FFD weight exceeds  $\frac{1}{3}$ .*

*Proof.* Suppose that  $B^*$  is a bin of the optimal packing that has weight greater than  $\frac{6}{5}$  and  $B^*$  contains neither of the items mentioned in the statement of the lemma. Clearly  $B^*$  must contain at least 3 items.

*Case 1.* Suppose  $|B^*| = 3$ . Then at least one item must have weight greater than  $\frac{1}{3}$  and, from the assumptions of the lemma, it can only have type  $X_2$ . There cannot be two such items, or else no item larger than  $l_{\text{last}}$  could fit with them. Thus  $w_F(B^*) \leq \frac{1}{2} + \frac{1}{3} + \frac{1}{3} = \frac{7}{6}$ .

TABLE 3  
Possible FFD weights for each nonexceptional item type.

Type	Weight
$Y_1$	$\frac{4}{5}, \frac{3}{4}, \frac{11}{15}, \frac{2}{3}, \frac{3}{5}$
$X_2$	$\frac{1}{2}, \frac{2}{5}$
$Y_2$	$\frac{2}{5}, \frac{11}{30}, \frac{1}{3}, \frac{4}{15}$
$X_3$	$\frac{1}{3}, \frac{4}{15}, \frac{1}{5}$
$Y_3$	$\frac{4}{15}, \frac{1}{5}$
$X_4$	$\frac{1}{4}, \frac{1}{5}$
$X_5$ or $Y_4$	$\frac{1}{5}$

*Case 2.* Suppose  $|B^*| = 4$ . If  $B^*$  does not contain an  $X_2$  item, then the smallest item packed must have weight exceeding  $\frac{1}{5}$ , else  $w_F(B^*) \leq 3(\frac{1}{3}) + \frac{1}{5} = \frac{6}{5}$ . No item larger than  $l_{\text{last}}$  can be packed with three items of type  $X_3$ , and there cannot be four items greater than  $\frac{1}{4}$  in size. Thus there must be one item of weight at most  $\frac{1}{4}$  and one other item of type  $Y_3$  or smaller, and  $w_F(B^*)$  is at most  $\frac{1}{3} + \frac{1}{3} + \frac{4}{15} + \frac{1}{4} < \frac{6}{5}$ . Consequently,  $B^*$  must contain an  $X_2$  item. The second largest item of  $B^*$  must be of type  $Y_3$  or  $X_4$ , implying that the remaining two items either (1) are each of type  $Y_4$  or less or (2) contain an item smaller than  $l_{\text{last}}$ . In either case,  $w_F(B^*) < \frac{6}{5}$ .

*Case 3.* Suppose  $|B^*| = 5$ .  $B^*$  must contain an  $X_3$  or  $Y_3$  item since it can contain neither a  $Y_2$  item nor four  $X_4$  items and any item as large as  $l_{\text{last}}$ . Therefore, the second largest item of  $B^*$  must be of type  $X_4$ , implying that the remaining three items either (1) are each of type  $Y_4$  or less or (2) contain an item smaller than  $l_{\text{last}}$ . In either case,  $w_F(B^*) < \frac{6}{5}$ .  $\square$

LEMMA 3.2. *The FFD packing of  $L$  contains no  $Y_2$  bin.*

*Proof.* Suppose there is a  $Y_2$  bin. Consider the sorted sublist  $L'$  obtained from  $L$  by deleting every item that is smaller than  $\frac{1}{6} + \Delta$ , every item that is larger than  $\frac{2}{3} - 2\Delta$ , and every item that is placed in a bin with an item larger than  $\frac{2}{3} - 2\Delta$  in the FFD packing of  $L$ . Clearly, the FFD packing of  $L'$  must also have a  $Y_2$  bin. Moreover, since  $\text{FFD}(L) > (\frac{6}{5}) \text{OPT}(L) + 8$ , it follows that  $\text{FFD}(L') > (\frac{6}{5}) \text{OPT}(L') + 8$ . (Deleting items smaller than  $\frac{1}{6} + \Delta$  does not affect the number of bins used by FFD and cannot increase the number required by OPT. After that, as long as the first item of the list is larger than  $\frac{2}{3} - 2\Delta$ , it and any other item FFD packs in  $B_1$  can be deleted, reducing the number of bins used by FFD by one and the number needed by OPT by at least one.) Thus, from these observations and the last lemma, it suffices to restrict our attention to  $L'$  and an optimal bin  $B^*$  that contains  $z$ , a  $Y_1$  item or a  $Y_2$  item whose FFD weight exceeds  $\frac{1}{3}$ , and show that, due to the presence of a  $Y_2$  bin,  $w_F(B^*) \leq \frac{6}{5}$ . We assume  $w_F(B^*) > \frac{6}{5}$  and consider the possible cases.

*Case 1.* Suppose  $z$  is a  $Y_1$  item.

Suppose  $w_F(z) > \frac{3}{5}$ . Then the smaller  $Y_2$  item in the  $Y_2$  bin did not fit with  $z$  in the FFD packing. Hence  $s(z) > 1 - (\frac{5}{12} - \Delta/2) = \frac{7}{12} + \Delta/2$ . If  $|B^*| = 2$ , then the second item can have weight at most  $\frac{1}{3}$  and since the weight of  $z$  is at most  $\frac{4}{5}$ ,  $w_F(B^*) < \frac{6}{5}$ . Since  $|B^*|$  must be less than 4, we must have  $|B^*| = 3$ . If the second largest item were at least  $\frac{1}{4}$  in size, no third item would fit. If both items are of type  $Y_4$  or  $X_5$ , then  $w_F(B^*) \leq \frac{4}{5} + 2(\frac{1}{5}) = \frac{6}{5}$ . Thus there must be an  $X_4$  item in  $B^*$  and, moreover, it must have weight  $\frac{1}{4}$ . If FFD packs this  $X_4$  item in a bin with subscript less than that of the bin containing  $z$ , then the  $Y_1$  weighting rule implies that its weight is at most  $1 - w_F(z)$  and we would get  $w_F(B^*) \leq \frac{6}{5}$ . But this  $X_4$  item would fit with  $z$ , so the item packed with  $z$  by the FFD algorithm is at least as large as an  $X_4$  item. Thus  $w_F(z) \leq \frac{3}{4}$  and  $w_F(B^*) \leq \frac{6}{5}$ . (Note that the  $Y_1$  weighting rule cannot cause  $z$  to have a weight exceeding  $\frac{3}{4}$  unless every  $X_4$  item has a weight less than  $\frac{1}{4}$ .)

Now suppose that  $w_F(z) = \frac{3}{5}$ . Then certainly  $|B^*| = 3$ . No item of size greater than  $\frac{1}{3}$  can then be used. If either of the other items had weight less than  $\frac{1}{3}$ , then  $w_F(B^*) \leq \frac{3}{5} + \frac{1}{3} + \frac{4}{15} = \frac{6}{5}$ . However, the only items of weight  $\frac{1}{3}$  have size greater than  $\frac{1}{4}$ , and no two items of size greater than  $\frac{1}{4}$  could fit with a  $Y_1$  item. We conclude that  $z$  cannot be a  $Y_1$  item.

*Case 2.* Suppose  $z$  is a  $Y_2$  item.

Clearly  $|B^*| = 3$  or 4. Suppose  $|B^*| = 3$ . The only possible problem occurs if  $B^*$  contains an  $X_2$  item,  $a$ , and an  $X_3$  item,  $b$ . In this event,  $\Delta > \frac{1}{30}$ , or else  $s(B^*) > 1$ . But then  $s(z) + s(b) > \frac{1}{3} + \frac{5}{18} - \Delta/3 > \frac{2}{3} - 2\Delta$ , the maximum size for a  $Y_1$  item. Thus  $a$  would fit with any  $Y_1$  item. Since it must be the case that  $w_F(a) = \frac{1}{2}$ , all fallback items

in  $Y_1$  bins must be of type  $X_2$ . Therefore  $z$  is packed by FFD into some  $Y_2$  bin,  $B_i$ . Certainly  $b$  would fit as the fallback item in  $B_i$ , and we conclude that either  $w_F(b) = \frac{4}{15}$  or  $w_F(z) = 11/30$ . In either case,  $w_F(B^*) \leq \frac{6}{5}$ .

Suppose  $|B^*| = 4$ . The second largest item of  $B^*$  can only be of type  $X_3$ , the third only of type  $X_4$ . Thus  $w(B^*) < \frac{6}{5}$  unless the smallest item is an  $X_4$  item as well. But this is impossible, since  $s(Y_2) + s(X_3) + 2s(X_4) \leq 1$  implies  $\Delta > \frac{1}{30}$  and  $s(Y_2) + s(X_3) + 2s(\text{any item}) \leq 1$  implies  $\Delta < \frac{1}{30}$ . We conclude that  $z$  cannot be a  $Y_2$  item.

By definition,  $w_F(L') \geq \text{FFD}(L') - 8$ . Lemma 3.1 and the analysis just completed demonstrate that  $w_F(L') \leq (\frac{6}{5}) \text{OPT}(L')$ . Hence we derive  $\text{FFD}(L') \leq (\frac{6}{5}) \text{OPT}(L') + 8$ , contradicting the presumed existence of a  $Y_2$  bin.  $\square$

We state here some important consequences that follow from our analysis of the FFD packing.

**COROLLARY 3.1.** *If  $x$  is a  $Y_2$  item, then  $w_F(x) \leq \frac{1}{3}$ . If  $B^*$  is any optimal bin not containing a  $Y_1$  item, then  $w_F(B^*) \leq \frac{6}{5}$ .*

**LEMMA 3.3.** *If  $B^*$  is any bin of the optimal packing containing an item of size less than  $\frac{1}{6} + \Delta$ , then  $w_F(B^*) \leq 1$ .*

*Proof.* Suppose  $B^*$  contains such an item,  $a$ . Then certainly  $|B^*|$  must be at least 3, since  $a$  is exceptional and therefore  $w_F(a) = 0$ .

*Case 1.* Suppose  $|B^*| = 3$ . Then there must be a  $Y_1$  item,  $b$ . The remaining item,  $c$ , would fit when  $b$  was packed. If it is unavailable, then its weight is at most  $1 - w_F(b)$  by the  $Y_1$  weighting rule. If it is available, then the item used in place of  $c$  must be at least as large. Since  $s(c) < \frac{1}{3}$ , there is no way for  $c$  to receive more weight than the item packed with  $b$  by FFD (see Tables 1, 2, and 3).

*Case 2.* Suppose  $|B^*| = 4$ . There must be an item of weight exceeding  $\frac{1}{3}$  that, by Lemma 3.2, cannot be of type  $Y_2$ . Thus it must be an  $X_2$  item. If each of the remaining items have weight at most  $\frac{1}{4}$ , then the lemma holds for  $B^*$ , so there must be a  $Y_3$  or  $X_3$  item. If both items have size at least  $\frac{1}{6} + \Delta$ , then  $s(B^*) > \frac{5}{12} - \Delta/2 + \frac{1}{4} + \frac{1}{6} + \frac{1}{6} + \Delta > 1$ . On the other hand, if there is a second item whose size is less than  $\frac{1}{6} + \Delta$ , then certainly  $w_F(B^*) \leq 1$ .

*Case 3.* Suppose  $|B^*| = 5$ . There must be an item of weight exceeding  $\frac{1}{4}$ , or else  $w_F(B^*) \leq 4(\frac{1}{4})$ . It cannot be larger than  $\frac{1}{3}$  in size, so it must be of type  $X_3$  or  $Y_3$ . There cannot be two items exceeding  $\frac{1}{4}$  in size, or else  $s(B^*) > 1$ . The remaining three items must all have size at least  $\frac{1}{6} + \Delta$ . If two are less than  $\frac{1}{5}$  in size, however,  $w_F(B^*) \leq \frac{1}{3} + \frac{1}{4} + 2(\frac{1}{5}) < 1$ . If two are to receive weight  $\frac{1}{4}$ , however,  $s(B^*) > \frac{1}{4} + \frac{5}{12} - \Delta/2 + \frac{1}{6} + \frac{1}{6} + \Delta > 1$ .

Thus, in any case, we conclude that  $w_F(B^*)$  is at most 1 if  $B^*$  contains an item smaller than  $l_{\text{last}}$ .  $\square$

**4. A close look at B2F.** We now seek to determine the precise conditions necessary for B2F( $L$ ) to exceed  $(\frac{6}{5}) \text{OPT}(L) + 8$ . In defining the weighting function  $w_B$  for the B2F packing, we shall retain the type classification described in § 2. That is, items are still classified strictly according to size as listed in Table 1. Most of our definition for  $w_B$  is straightforward and is given in Table 4.

The definition of  $w_B$  for items in  $Y_1$  bins is more complicated and is described in the following paragraphs.

We wish to maintain the fact that the sum of the weights of the items in any nonexceptional bin is 1. Thus in any  $Y_1$  bin with only one item, that item has weight 1. (Unlike the FFD packing, such a one-item bin may exist in the B2F packing.) We would also like to keep smaller  $Y_1$  items from having greater weight than larger ones, and we would like the fallback items to have their weight assigned according to their type. The difficulty

TABLE 4  
Weighting function  $w_B$  for bins not containing an item  
of size exceeding  $\frac{1}{2}$  in B2F packing.

Type of nonexceptional items in a B2F-packed bin	Weights assigned
$X_2$ or $Y_2$ , $X_2$ or $Y_2$	$\frac{1}{2}, \frac{1}{2}$
$X_2$ or $Y_2$ , $X_2$ or $Y_2$ , any item	$\frac{2}{5}, \frac{2}{5}, \frac{1}{5}$
$X_2$ or $Y_2$ , $X_3$ or $Y_3$ , $X_3$ or $Y_3$	$\frac{2}{5}, \frac{3}{10}, \frac{3}{10}$
$X_2$ or $Y_2$ , $X_3$ or $Y_3$ , $X_4$ or smaller item	$\frac{1}{2}, \frac{3}{10}, \frac{1}{5}$
$X_2$ or $Y_2$ , $X_4$ or $Y_4$ , $X_4$ or $Y_4$	$\frac{1}{2}, \frac{1}{4}, \frac{1}{4}$
$X_3$ or $Y_3$ , $X_3$ or $Y_3$ , $X_3$ or $Y_3$	$\frac{1}{3}, \frac{1}{3}, \frac{1}{3}$
$X_3$ or $Y_3$ , $X_3$ or $Y_3$ , $X_3$ or $Y_3$ , $X_4$ or smaller item	$\frac{4}{15}, \frac{4}{15}, \frac{4}{15}, \frac{1}{5}$
$X_3$ or $Y_3$ , $X_3$ or $Y_3$ , $X_4$ or smaller item, $X_4$ or smaller item	$\frac{3}{10}, \frac{3}{10}, \frac{1}{5}, \frac{1}{5}$
$X_4$ or $Y_4$ , $X_4$ or $Y_4$ , $X_4$ or $Y_4$ , $X_4$ or $Y_4$	$\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}$
any five items	$\frac{1}{5}, \frac{1}{5}, \frac{1}{5}, \frac{1}{5}, \frac{1}{5}$

comes with small items (those with size less than or equal to  $\frac{1}{3}$ ), in which case  $w_B$  depends on the last such item packed in a  $Y_1$  bin.

Specifically, let  $h$  be the index of the last bin in the B2F packing containing a  $Y_1$  item, no  $X_2$  or  $Y_2$  item, and at most one fallback item. All subsequent  $Y_1$  bins contain either two fallback items or one fallback item of type  $Y_2$  or  $X_2$ . In either case, the  $Y_1$  item is given weight  $\frac{3}{5}$ . If there is one fallback item, it is given weight  $\frac{2}{5}$ ; if there are two, each is given weight  $\frac{1}{5}$ .

If  $|B_h| = 1$ , then a  $B_h$ 's  $Y_1$  item and all earlier  $Y_1$  items are assigned weight 1, and all earlier fallback items are assigned weight zero.

If  $B_h = \{y, x\}$ , where  $y$  is of type  $Y_1$  and  $s(x) \leq \frac{1}{3}$ , then we determine the weight of  $x$  by examining all items of size less than or equal to  $s(x)$  that are packed after the last  $Y_1$  item. That is, we set  $w_B(x) = \max \{w_B(t) \mid s(t) \leq s(x), t \text{ not packed in a } Y_1 \text{ bin}\}$ . Of all items that are available when  $x$  is packed that would fit (no larger item would fit), and that are not packed in  $Y_1$  bins, we choose the one that has maximum weight (using Table 4). If there are no such items, then we set  $w_B(x) = \text{zero}$ .

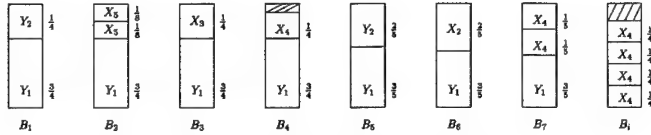
Once  $B_h$  and  $w_B(x)$  have been determined, the rest of  $w_B$  is defined as follows. The  $Y_1$  item  $y$  in  $B_h$  is given weight  $w_B(y) = 1 - w_B(x)$ . Since  $s(x) \leq \frac{1}{3}$  and the maximum size of any  $Y_1$  item is  $\frac{2}{3}$ ,  $x$  must have fit in any preceding bin. Thus each such bin contains either two fallback items, or one fallback item at least as large as  $x$ . All  $Y_1$  items preceding  $B_h$  are assigned weight  $w_B(y)$ . If there are two fallback items, each is assigned weight  $w_B(x)/2$ ; if there is only one, it is assigned weight  $w_B(x)$ .

If  $B_h$  does not exist, then  $h = 0$  and all  $Y_1$  items are assigned weight  $\frac{3}{5}$  with their associated fallback items given weight  $\frac{2}{5}$ , or  $\frac{1}{5}$  each if there are two of them.

The example depicted in Fig. 2 illustrates the role of  $B_h$  in determining  $w_B$ . Types of items packed in each bin are given on the inside,  $w_B$  is listed on the outside. In this example,  $h = 4$ , and one of the  $X_4$  items in  $B_i$  is no larger than the  $X_4$  item in  $B_4$ .

**DEFINITION.** The following bins are exceptional for the B2F packing: the last bin to contain an item of each of the types  $X_2$ ,  $Y_2$ ,  $X_3$ ,  $Y_3$ ,  $X_4$ ,  $Y_4$ , the last bin containing exactly three  $X_3$  or  $Y_3$  items, and the last bin of the packing.

In general, therefore, the last bin containing an item of a particular type is exceptional, although  $Y_1$  and  $X_5$  items are excluded from this. Note that if an  $X_2$  item is packed with two  $Y_4$  items, there can be no  $X_2$  items left (since any  $X_2$  item is larger than any two  $Y_4$

FIG. 2. The role of  $B_h$  in determining  $w_B$ . Here,  $h = 4$ .

items) and the bin is exceptional. If an  $X_2$  item is packed with an  $X_4$  item and a  $Y_4$  item, there can be no  $X_4$  items left and the bin is exceptional. Similarly, if an  $X_2$  or  $Y_2$  item is packed with an  $X_4$  or  $Y_4$  item and an  $X_5$  item, the bin is exceptional since there can be no more  $X_4$  or  $Y_4$  items available. A bin whose largest item is of type  $X_3$  ( $Y_3$ ) is configured as described in Table 4 unless there are no more  $X_3$  ( $Y_3$ ) items available. Also, a bin whose largest item is of type  $X_4$  ( $Y_4$ ) is configured as described in Table 4 unless there are no more  $X_4$  ( $Y_4$ ) items left. Finally, the last bin containing three  $X_3$  or  $Y_3$  items and nothing else is classified as exceptional. Although this bin might not otherwise qualify as exceptional, it could cause problems in our proof if its items were each to receive weight  $\frac{1}{3}$ .

We conclude that there are at most eight exceptional bins in the B2F packing. We define an exceptional item simply as one packed in an exceptional bin. Such an item receives a weight of zero, completing our definition of  $w_B$ . For the convenience of the reader, Table 5 provides a listing of the possible weights for each nonexceptional item type.

Before proceeding with the principle results of this section, we first prove some preliminary lemmas that reveal details of the B2F packing. The first of these concerns is the occurrence of items of weight  $\frac{1}{3}$ , the second the impossibility of a certain configuration containing  $Y_3$  and  $Y_4$  items.

**LEMMA 4.1.** *If there is an item,  $x$ , of B2F weight  $\frac{1}{3}$ , then there must be a bin in the B2F packing containing exactly three items, each of which has size no larger than  $s(x)$ .*

*Proof.* The only possible types for  $x$  are  $X_3$  and  $Y_3$ , and the only possible bins for  $x$  to be packed in are a three-item bin or one with an item of type  $Y_1$ , packed in a bin  $B_i$ , where  $i \leq h$ . Suppose  $x$  is packed with a  $Y_1$  item. From the definition of  $w_B$ , it is clear that the fallback item in  $B_h$  also has weight  $\frac{1}{3}$  and is no larger than  $x$ . Without loss of generality, we can assume that  $x$  is the fallback item in  $B_h$ . If  $x$  has weight  $\frac{1}{3}$ , however, then there must be another item packed after the  $Y_1$  bins that is no larger than  $x$  and

TABLE 5  
Possible B2F weights for nonexceptional  
items in a bin  $B_i$ , where  $i$  exceeds  $h$ .

Type	Weight
$Y_1$	$\frac{3}{5}$
$X_2$	$\frac{1}{2}, \frac{2}{5}$
$Y_2$	$\frac{1}{2}, \frac{2}{5}$
$X_3$	$\frac{1}{3}, \frac{3}{10}, \frac{4}{15}, \frac{1}{5}$
$Y_3$	$\frac{1}{3}, \frac{3}{10}, \frac{4}{15}, \frac{1}{5}$
$X_4$	$\frac{1}{4}, \frac{1}{5}$
$Y_4$	$\frac{1}{4}, \frac{1}{5}$
$X_5$	$\frac{1}{5}$

has weight  $\frac{1}{3}$ . From Table 4 we know that this item must be packed in a bin containing exactly three items each of weight  $\frac{1}{3}$ . Moreover, we know from Table 1 that any one of these three items would have been used in place of  $x$  if it were larger. Thus we may assume that the lemma holds unless  $x$  is an  $X_3$  or  $Y_3$  item packed in a three-item bin. However, the last such bin will contain the three smallest such items (and hence is exceptional). Thus the last three-item bin satisfies the conditions of the lemma.  $\square$

LEMMA 4.2. *If there is a  $Y_4$  item of B2F weight  $\frac{1}{4}$ , then there is no  $Y_3$  item of B2F weight  $\frac{1}{3}$ .*

*Proof.* Suppose there are such  $Y_3$  and  $Y_4$  items. In order to have a  $Y_3$  item of weight  $\frac{1}{3}$ , there must be a B2F bin,  $B$ , containing three  $Y_3$  items and nothing else. (A bin with three items, some of type  $X_3$  and some of type  $Y_3$ , would be exceptional since it would contain the last  $X_3$  item, and hence its items would have weight zero.) Of course, a  $Y_3$  item can have weight  $\frac{1}{3}$  if it is packed in a  $Y_1$  bin, but even in this case there must be another bin containing three  $Y_3$  items of weight  $\frac{1}{3}$ . If there were two  $Y_4$  items available when  $B$  was packed, then they would have replaced the last  $Y_3$  item since any two  $Y_3$  items and any two  $Y_4$  items will always fit in a single bin. Thus the  $Y_4$  items must have been packed as fallback items in a bin before  $B$  was packed. The only way such a fallback item can have weight  $\frac{1}{4}$  is if it is packed in a bin containing a  $Y_2$  item and two  $Y_4$  items. (Note that a bin containing an  $X_2$  item and two  $Y_4$  items, or one  $X_4$  item and one  $Y_4$  item, is exceptional.) But if such a bin were to occur before  $B$ , then two of the available  $Y_3$  items would have been packed instead with the  $Y_2$  item. Thus we cannot have both items, as specified in the statement of the lemma.  $\square$

LEMMA 4.3. *If  $B^*$  is a bin of the optimal packing containing a  $Y_1$  item, then  $w_B(B^*) \leq \frac{6}{5}$ .*

*Proof.* Assume otherwise for some bin  $B^*$  containing a  $Y_1$  item,  $a$ . Since  $a$  cannot fit with three or more items in any bin, we must have  $|B^*| \leq 3$ . We begin by observing that if  $a$  has weight exceeding  $\frac{3}{5}$ , then we can, without loss of generality, assume that  $a$  is the  $Y_1$  item B2F packed in  $B_h$ . Otherwise, it would come from a bin preceding  $B_h$  in the B2F packing, and would consequently be at least as large as the  $Y_1$  item in  $B_h$ . Thus the  $Y_1$  item in  $B_h$  would fit in  $B^*$  in place of  $a$ , and we may as well assume that it is  $a$ .

*Case 1.* Suppose  $|B^*| = 2$ . Then certainly some item in  $B^*$  must have weight exceeding  $\frac{3}{5}$ , and we can assume that  $a$  is packed in  $B_h$ . Let  $b$  be the other item in  $B^*$ . Since  $b$  would fit in  $B_h$ , either  $b$  was packed earlier and thus was not available, or the item packed with  $a$  in  $B_h$  is at least as large as  $b$ . If  $b$  is packed by B2F in a  $Y_1$  bin after  $B_h$ , then  $w_B(b) = \frac{1}{5}$  since  $b$  cannot be of type  $X_2$  or  $Y_2$  (if it were, the item packed in  $B_h$  could not be of size  $\frac{1}{3}$  or less). Thus  $w_B(B^*) \leq \frac{6}{5}$  in this case. If  $b$  is packed before  $a$ , or if  $b$  is packed after the  $Y_1$  bins,  $w_B(b) \leq 1 - w_B(a)$  and so  $w_B(B^*)$  is at most 1 in this case. We conclude that if  $|B^*| = 2$ ,  $w_B(B^*)$  cannot exceed  $\frac{6}{5}$ .

*Case 2.* Suppose  $|B^*| = 3$ . Let  $B^* = \{a, b, c\}$  with  $s(b) \geq s(c)$ . Then  $s(b) < \frac{1}{3}$  and  $s(c) < \frac{1}{4}$ , or else  $s(B^*) > 1$ . Therefore, their weights are at most  $\frac{1}{3}$  and  $\frac{1}{4}$ , respectively. Consequently, we know that  $w_B(a)$  must exceed  $\frac{3}{5}$  if the lemma is to fail. Thus we can assume that  $a$  is the  $Y_1$  item in  $B_h$ . We now employ the same argument that we used in Case 1 to prove that the sum of the weights of  $a$  and either  $b$  or  $c$  can be at most 1. If both were available, then  $B_h$  would use two fallback items, so either  $b$  or  $c$  must be packed before  $a$ . Then certainly the sum of the weight of  $a$  and the weight of that item is at most 1. If one is still available, and it is not packed in a  $Y_1$  bin after  $a$ , then it is no larger than the item packed with  $a$  by B2F. Consequently, its weight is no greater, and the sum of its weight and that of  $a$  is at most 1. If the available item is packed in a  $Y_1$  bin after  $B_h$ , then its weight cannot be  $\frac{2}{5}$  since its size is at most  $\frac{1}{3}$ . But if its weight is  $\frac{1}{5}$ , the weight of

$B^*$  is at most  $\frac{6}{5}$ . From this we conclude that both  $b$  and  $c$  must have weight exceeding  $\frac{1}{5}$ .

Let  $d$  be the fallback item in  $B_h$ . Thus  $s(b) + s(c) > s(d)$ , because  $s(d) \leq \frac{1}{3}$  and  $s(b) \geq s(c) > \frac{1}{6}$ . It could not be the case that both  $b$  and  $c$  were available when  $d$  was packed, or else they would have replaced  $d$ . Suppose  $s(d) > s(b)$ . Since  $s(d) + s(\text{any } Y_1 \text{ item}) \leq 1$ , and since  $d$  is larger than either  $b$  or  $c$ , whichever of these is packed before  $d$  must be one of two fallback items in its bin and hence will have weight less than  $\frac{1}{5}$ . Suppose  $s(b) \geq s(d) > s(c)$ . Then  $c$  would have fit with  $a$  and  $d$  in  $B_h$  had it been available. Since it was not used, we conclude that  $c$  must be packed before  $d$  in a bin with two fallback items, and hence has weight less than  $\frac{1}{5}$ . The only remaining possibility is that  $s(c) \geq s(d)$ . Now, however, any item no larger than  $d$  would fit in  $B_h$  with  $a$  and  $d$ . Since none was placed there, none can have been left to be packed after the  $Y_1$  bins, and therefore  $w_B(d)$  is zero. In this event, since  $b$  and  $c$  are packed before  $d$ ,  $w_B(b)$  and  $w_B(c)$  are zero as well, contradicting the assumption that  $w_B(B^*) > \frac{6}{5}$ .  $\square$

LEMMA 4.4. *The B2F weight of an optimal bin cannot exceed  $\frac{6}{5}$  unless the bin contains either a  $Y_2$  item of B2F weight greater than  $\frac{1}{3}$  or an item of size less than  $\frac{1}{6} + \Delta$ .*

*Proof.* To obtain the proof, suppose otherwise for some  $B^*$ . We know from Lemma 4.3 that  $B^*$  cannot contain a  $Y_1$  item. It is easy to see then that  $|B^*| \geq 3$ .

*Case 1.* Suppose  $|B^*| = 3$ . Then the only item of weight exceeding  $\frac{1}{3}$  can be an  $X_2$  item. Since any two such items and an item of size greater than  $\frac{1}{6} + \Delta$  would be too big to fit, there can be at most one item of weight exceeding  $\frac{1}{3}$  and  $w_B(B^*) \leq \frac{1}{2} + 2(1/3) < \frac{6}{5}$ .

*Case 2.* Suppose  $|B^*| = 4$ . Suppose first that the largest item in  $B^*$  is an  $X_2$  item. There cannot be another item of size greater than  $\frac{1}{4}$ , because then the sum of these sizes would exceed  $\frac{5}{12} - \Delta/2 + \frac{1}{4} + 2(\frac{1}{6} + \Delta) > 1$ . Items of size at most  $\frac{1}{4}$  ( $X_4, Y_4, X_5$ ) can have weight at most  $\frac{1}{4}$ . If any of these items were to have weight less than or equal to  $\frac{1}{5}$ , then  $w_B(B^*)$  would be at most  $\frac{1}{2} + 2(\frac{1}{4}) + \frac{1}{5} = \frac{6}{5}$ . Thus all three items besides the  $X_2$  item must be  $X_4$  or  $Y_4$  items of weight  $\frac{1}{4}$ . But such items have size exceeding  $\frac{1}{5}$  and then  $s(B^*) > \frac{5}{12} - \Delta + 3(\frac{1}{5})$  which is at least 1 if  $\Delta \leq \frac{1}{30}$ . If  $\Delta > \frac{1}{30}$ , however,  $s(B^*) > \frac{5}{12} - \Delta/2 + 3(\frac{1}{6} + \Delta) > 1$ . Thus in all cases where  $|B^*| = 4$  and  $B^*$  contains an  $X_2$  item,  $w_B(B^*) \leq \frac{6}{5}$ .

Suppose now that the largest item is a  $Y_2$  item, which has weight less than or equal to  $\frac{1}{3}$  by assumption. If there were two additional items of size greater than  $\frac{1}{4}$ , we would have  $s(B^*) > \frac{1}{3} + 2(\frac{1}{4}) + \frac{1}{6} + \Delta > 1$ . Thus there must be two items of size less than or equal to  $\frac{1}{4}$ , and hence of weight at most  $\frac{1}{4}$ . Since there can be no item of weight exceeding  $\frac{1}{3}$ , we must have  $w_B(B^*) \leq 2(\frac{1}{3} + \frac{1}{4}) < \frac{6}{5}$ .

Since  $|B^*| = 4$ , there must be at least one item of weight exceeding  $\frac{3}{10}$ , which must be of type  $X_3$  or  $Y_3$  and of weight  $\frac{1}{3}$ . If any item has weight less than or equal to  $\frac{1}{5}$ ,  $w_B(B^*)$  would be at most  $3(\frac{1}{3}) + \frac{1}{5} = \frac{6}{5}$ . If there are two items of weight  $\frac{1}{4}$ , we would still have  $w_B(B^*) < \frac{6}{5}$ . There cannot be four items of size greater than  $\frac{1}{4}$ , so there must be an  $X_4$  or  $Y_4$  item of weight  $\frac{1}{4}$  and three  $X_3$  or  $Y_3$  items. At least two of the  $X_3$  or  $Y_3$  items must have weight  $\frac{1}{3}$ , and so there must be a bin in the B2F packing containing three  $X_3$  or  $Y_3$  items of weight  $\frac{1}{3}$ . In particular, the last three-item bin is exceptional and must contain three items no larger than those in  $B^*$ . (Even if the items in  $B^*$  are fallback items, there must be such a three-item bin, and the last bin is exceptional.) If the item,  $x$ , of weight  $\frac{1}{4}$  were still available when this three-item bin was packed, then  $x$  and any other item of weight  $\frac{1}{4}$  would replace the bin's last item. Thus  $x$  must be packed as a fallback item in an earlier bin. The only way to have weight  $\frac{1}{4}$  would be in a bin with an  $X_2$  (or  $Y_2$ ) item and another item of weight  $\frac{1}{4}$ . In this event, however,  $x$  and any of the items in the last

three-item bin would fit with the  $X_2$  item since they fit with two other  $X_3$  or  $Y_3$  items. (Note that  $x$  cannot be packed as a single fallback item in a  $Y_1$  bin, since any  $X_3$  or  $Y_3$  item would fit and be used instead of  $x$ .)

*Case 3.* Suppose  $|B^*| = 5$ . If any item has size exceeding  $\frac{1}{3}$ ,  $s(B^*)$  would be greater than  $\frac{1}{3} + 4(\frac{1}{6} + \Delta) > 1$ . Similarly, not all items can have size exceeding  $\frac{1}{5}$ . If all items are no larger than  $\frac{1}{4}$  in size, then the weight of  $B^*$  would be at most  $\frac{6}{5}$  since none of the items could have weight more than  $\frac{1}{4}$  and at least one would have weight at most  $\frac{1}{5}$ . Thus there must be at least one  $X_3$  or  $Y_3$  item, and at least one  $X_5$  item.

Suppose there is an  $X_3$  item. If there are two additional items of size greater than  $\frac{1}{5}$ ,  $s(B^*) > \frac{5}{18} - \Delta/3 + 2(\frac{1}{5}) + 2(\frac{1}{6} + \Delta) > 1$ . Thus there is at most one item of weight exceeding  $\frac{1}{4}$  and one additional item of size exceeding  $\frac{1}{5}$ . Hence,  $w_B(B^*) \leq \frac{1}{3} + \frac{1}{4} + 3(\frac{1}{5}) < \frac{6}{5}$ .

Suppose finally that the largest item in  $B^*$  is of type  $Y_3$ .  $B^*$  can contain at most one such item, or else  $s(B^*) > 2(\frac{1}{4}) + 3(\frac{1}{6} + \Delta) > 1$ . Also,  $B^*$  cannot contain a  $Y_3$  item and two  $X_4$  items, or else  $s(B^*) > \frac{1}{4} + 2(\frac{5}{24} - \Delta/4) + 2(\frac{1}{6} + \Delta) > 1$ . However, if  $B^*$  contains three items each of weight less than or equal to  $\frac{1}{5}$ ,  $w_B(B^*) \leq \frac{1}{3} + \frac{1}{4} + 3(\frac{1}{5}) < \frac{6}{5}$ . There must be at least two items of size (and hence weight) no greater than  $\frac{1}{5}$ , or else  $s(B^*) > \frac{1}{4} + 3(\frac{1}{5}) + \frac{1}{6} + \Delta > 1$ . The only remaining possibility is for  $B^*$  to contain a  $Y_3$  item, a  $Y_4$  item, an  $X_4$  or  $Y_4$  item, and two  $X_5$  items. By Lemma 4.2, either the  $Y_3$  item has weight less than  $\frac{1}{3}$  or the  $Y_4$  item has weight less than  $\frac{1}{4}$ . Either of these possibilities contradicts the assumption that  $w_B(B^*) > \frac{6}{5}$ .  $\square$

LEMMA 4.5. *There cannot be a  $Y_1$  item  $a$  with  $w_B(a) \geq \frac{4}{3}$  if there exist items  $b$  and  $c$  with  $w_B(b) = \frac{1}{3}$ ,  $s(c) > \max(\frac{5}{24} - \Delta/4, \frac{1}{6} + \Delta)$ , and  $s(a) + s(b) + s(c) \leq 1$ .*

*Proof.* We shall show that under these conditions no bin of the optimal packing can have a B2F weight exceeding  $\frac{6}{5}$ . Suppose  $L$  contains such items and that, for some optimal bin  $B^*$ ,  $w_B(B^*) > \frac{6}{5}$ . As we have seen before, there is no loss of generality in assuming that  $a$  is the  $Y_1$  item in  $B_h$ . We know from Lemma 4.4 that  $B^*$  must contain a  $Y_2$  item of weight greater than  $\frac{1}{3}$  or an item of size less than  $\frac{1}{6} + \Delta$ . If there is a  $Y_2$  item of weight exceeding  $\frac{1}{3}$ , however, then there must have been such an item available when  $B_h$  was packed. Since any such item is smaller than the sum of the sizes of  $b$  and  $c$ , it would fit with  $a$  in  $B_h$ , contradicting the definition of  $B_h$ . Thus the only possibility is for  $B^*$  to contain an item,  $d$ , of size less than  $\frac{1}{6} + \Delta$ . When  $a$  was packed, if  $d$  and any other item of size at most that of  $b$  were available, they would be used in place of the fallback item in  $B_h$ . Since  $w_B(b) = \frac{1}{3}$ , either  $b$  itself must have been available or  $b$  must be packed in an earlier  $Y_1$  bin and some item no larger than  $b$  must have been available. In either case, there must have been an item no larger than  $b$  available when  $B_h$  was packed. Thus  $d$  must not have been available. But if  $d$  is packed in a  $Y_1$  bin before  $B_h$ , it cannot be the only fallback item, since there must be an item of weight  $\frac{1}{3}$ , no larger than  $b$ , available that would fit with any  $Y_1$  item. Thus  $d$  must have weight no greater than  $\frac{1}{6}$ . At this point, we must consider the possible configurations for  $B^*$ . Certainly  $B^*$  must contain at least three items.

*Case 1.* Suppose  $|B^*| = 3$ . By Lemma 4.3,  $B^*$  cannot contain an item of weight greater than  $\frac{1}{2}$ . Thus  $w_B(B^*) \leq \frac{1}{2} + \frac{1}{2} + \frac{1}{6} = \frac{7}{6}$ .

*Case 2.* Suppose  $|B^*| = 4$ .  $B^*$  must contain an item of weight greater than  $\frac{1}{3}$ , which can only be an  $X_2$  item by Lemma 4.3 and by the above arguments focusing on the weight of  $Y_2$  items. Moreover, if there is not a second item of weight greater than  $\frac{1}{4}$ , then we would have  $w_B(B^*) \leq \frac{1}{2} + \frac{1}{4} + \frac{1}{4} + \frac{1}{6} = \frac{7}{6}$ . If there is a second item larger than  $\frac{1}{4}$  in size, then there must be a second item whose size is less than  $\frac{1}{6} + \Delta$ , or else  $s(B^*) > \frac{5}{12} - \Delta/2 + \frac{1}{4} + \frac{1}{6} + \frac{1}{6} + \Delta > 1$ . Since this second small item will also have weight no greater than  $\frac{1}{6}$ , we again have that  $w_B(B^*) \leq \frac{1}{2} + \frac{1}{3} + 2(\frac{1}{6}) = \frac{7}{6}$ .

*Case 3.* Suppose  $|B^*| = 5$ .  $B^*$  must contain an item of weight greater than  $\frac{1}{4}$ . There cannot be two such items, or else  $s(B^*) > 1$ . If any of the remaining items has weight less than or equal to  $\frac{1}{5}$ , then  $w_B(B^*) \leq \frac{1}{3} + 2(\frac{1}{4}) + \frac{1}{5} + \frac{1}{6} = \frac{6}{5}$ . But this means that there must be one item whose size exceeds  $\frac{1}{4}$  and three additional items each of whose sizes exceeds  $\frac{1}{5}$ . Thus  $s(B^*) > \frac{1}{4} + 3(\frac{1}{5}) + \frac{1}{6} > 1$ . Hence, in all cases,  $w_B(B^*) \leq \frac{6}{5}$ .

To complete the proof of Lemma 4.5, we observe that  $w_B(L) \geq \text{B2F}(L) - 8$  since each nonexceptional bin has a weight of 1, while  $w_B(L) \leq (\frac{6}{5}) \text{OPT}(L)$  since each optimal bin has a weight bounded above by  $\frac{6}{5}$ . Combining these yields  $\text{B2F}(L) \leq (\frac{6}{5}) \text{OPT}(L) + 8$ , contradicting the assumption that  $L$  was a counterexample for CFB.  $\square$

**5. Proof of the main result.** We shall now employ our weighting function averaging technique to obtain the final result. From Corollary 3.1 and Lemma 4.4 we know the optimal bin configurations that may have "too much" weight from the respective FFD or B2F weighting function, and that since FFD fails to achieve the required bound, any  $Y_2$  item receives an FFD weight of  $\frac{1}{3}$ . Also, from Lemma 4.5, we know that since B2F fails, any  $Y_1$  item either cannot be packed extremely well or receives a B2F weight of  $\frac{2}{3}$ . The heart of the proof of the main result is now contained in the following lemma.

**LEMMA 5.1.** *If  $B^*$  is any bin of the optimal packing of  $L$ ,  $w_A(B^*) = (w_F(B^*) + w_B(B^*))/2 \leq \frac{6}{5}$ .*

*Proof.* To obtain the proof, suppose otherwise for some optimal bin  $B^*$ . Clearly, at least one of the two weighting functions must give  $B^*$  a weight exceeding  $\frac{6}{5}$ .

*Case 1.* Suppose  $w_F(B^*) > \frac{6}{5}$ . Then we know that  $B^*$  must contain a  $Y_1$  item,  $a$ , and that  $|B^*| \leq 3$ . If  $|B^*| = 2$ , then the second item,  $b$ , would fit with  $a$  when  $a$  was packed. If it is unavailable, then the  $Y_1$  packing rule for FFD implies that  $b$  cannot have weight exceeding  $1 - w_F(a)$ . If  $b$  is available, then the item packed with  $a$  is at least as large as  $b$ . If  $b$  has weight less than or equal to  $\frac{2}{5}$ , then  $w_F(a) + w_F(b) \leq \frac{6}{5}$ , since  $a$  cannot have weight exceeding  $\frac{4}{5}$  unless nothing fits with it. Since we already know that there are no  $Y_2$  bins in the FFD packing by Lemma 3.2,  $b$  must be an  $X_2$  item. In this case, however,  $a$  must also be packed by FFD with an  $X_2$  item. Thus  $w_F(a) = \frac{3}{5}$  and  $w_F(B^*) < \frac{6}{5}$ .

Therefore, we may assume that  $B^* = \{a, b, c\}$ , where  $s(a) > s(b) \geq s(c)$ . It is easy to see that  $s(c) < \frac{1}{4}$  and  $s(b) < \frac{1}{3}$ , or else  $s(B^*)$  would exceed 1. Hence  $w_F(c) \leq \frac{1}{4}$  and  $w_F(b) \leq \frac{1}{3}$ , implying that  $w_F(a)$  must be greater than  $\frac{2}{5}$ . Let  $B_i$  denote the FFD bin containing  $a$ . Since  $b$  would fit in  $B_i$  with  $a$  (or any other  $Y_1$  item),  $w_F(b) \leq 1 - w_F(a)$ , and  $w_F(B^*) \leq \frac{6}{5}$ . Note further that  $c$  must be an  $X_4$  item, or else its weight would be  $\frac{1}{5}$  and  $B^*$  would have weight less than or equal to  $\frac{6}{5}$ .

This is, for those readers already acquainted with FFD, exactly the kind of situation where one expects FFD to perform poorly. We now show that, in this case, the averaging process with B2F permits our compound algorithm to succeed.

Suppose  $w_B(a) = \frac{3}{5}$ . Unless  $w_B(b) = \frac{1}{3}$  and  $w_B(c) = \frac{1}{4}$ , we have  $w_A(B^*) \leq (\frac{5}{4} + 23/20)/2 = \frac{6}{5}$ . Now  $w_B(b) = \frac{1}{3}$  implies the existence of a bin containing three items of type  $X_3$  or  $Y_3$ , each of weight  $\frac{1}{3}$ . There must also be a bin containing three such items each no larger than  $b$ , although their weight may be zero if they are exceptional. If  $c$  were available when this three-item bin was packed, it and any smaller item would replace the last  $Y_3$  or  $X_3$  item. But if  $c$  is the smallest item left, it is either  $l_{\text{last}}$  and hence is exceptional, or it is a fallback item and has weight at most  $\frac{1}{5}$ . Therefore,  $c$  must not be available. If  $c$  were packed in a  $Y_1$  bin, the items of the three-item bin would have been used unless  $c$  is packed with a second fallback item. However, it then has weight at most  $\frac{1}{5}$ . The only remaining possibility would be for  $c$  to be packed with another  $X_4$  or  $Y_4$  item

and an  $X_2$  or  $Y_2$  item. In this event, however,  $c$  and any item from the three-item bin would have fit with any  $X_2$  or  $Y_2$  item and been used instead. Thus we know that it is impossible for  $a$  to have weight  $\frac{2}{3}$  in the B2F weighting.

Suppose  $w_B(a) > \frac{2}{3}$ . It must be that  $a$  is in a  $Y_1$  bin packed no later than  $B_h$ , and as argued before there is no loss of generality in assuming that  $a$  is packed in  $B_h$ . Let  $d$  be the fallback item packed with  $a$  in  $B_h$ . Thus  $s(b) + s(c) > s(d)$ , and both  $b$  and  $c$  could not have been available when  $d$  was packed, else they would have replaced  $d$ . If  $s(d) > s(b)$ , then  $w_B(d) \geq \max\{w_B(b), w_B(c)\}$  and whichever of  $b$  or  $c$  is not available has weight less than or equal to  $(\frac{1}{2})w_B(d)$ . This causes  $w_B(B^*)$  to be at most  $1 + (\frac{1}{2})w_B(b)$  no matter where  $b$  was packed by B2F. Unless  $b$  has weight  $\frac{1}{3}$ , this quantity is at most  $23/20$  and  $w_A(B^*)$  would be at most  $\frac{6}{5}$ . But this is precisely the situation ruled out by Lemma 4.5. If  $s(b) \geq s(d) > s(c)$ , then we reach the same conclusion, since it must still be that  $w_B(d) \geq w_B(b)$  and since  $c$  is not available implying  $w_B(c) \leq (\frac{1}{2})w_B(d)$ . Finally, if  $s(c) \geq s(d)$ , then any item no larger than  $d$  would fit in  $B_h$  along with  $a$  and  $d$ . Since none was used,  $w_B(b)$ ,  $w_B(c)$  and  $w_B(d)$  are all zero.

Case 2. Suppose  $w_B(B^*) > \frac{6}{5}$  and  $B^*$  contains a  $Y_2$  item of B2F weight exceeding  $\frac{1}{3}$ .

Certainly  $|B^*| \leq 4$ , since no bin can contain an item of size greater than  $\frac{1}{3}$  and four additional items.

Suppose  $|B^*| = 4$ . Then there can be no other  $X_2$  or  $Y_2$  items and at most one other item of size exceeding  $\frac{1}{4}$ , or else  $s(B^*) > 1$ . If all other items are at most  $\frac{1}{4}$  in size, then  $w_B(B^*) \leq \frac{1}{2} + 3(\frac{1}{4}) = \frac{5}{4}$ . Since  $w_F(B^*) \leq \frac{1}{3} + 3(\frac{1}{4})$ , we would have  $w_A(B^*) < \frac{6}{5}$ . Therefore there must be an  $X_3$  or  $Y_3$  item.

Suppose  $B^*$  contains an  $X_3$  item. Then there must also be either an item of size at most  $\frac{1}{5}$  or an item of size less than  $\frac{1}{6} + \Delta$ . To see this, observe that if all items have size exceeding  $\frac{1}{5}$ ,  $s(B^*) > \frac{1}{3} + \frac{5}{18} - \Delta/3 + \frac{2}{5} \geq 1$  if  $\Delta \leq \frac{1}{30}$ . If all items have size greater than or equal to  $\frac{1}{6} + \Delta$ ,  $s(B^*) > \frac{1}{3} + \frac{5}{18} - \Delta/3 + 2(\frac{1}{6} + \Delta) \geq 1$  if  $\Delta > \frac{1}{30}$ . However, if there is an item of size less than  $\frac{1}{6} + \Delta$ , then  $w_F(B^*) \leq 2(\frac{1}{3}) + \frac{1}{4} + 0 = 11/12$  while  $w_B(B^*) \leq \frac{1}{2} + \frac{1}{3} + 2(\frac{1}{4}) = \frac{4}{3}$ . If, on the other hand,  $B^*$  contains an  $X_5$  item, then  $w_B(B^*) \leq \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} = 77/60$  while  $w_F(B^*) \leq 2(\frac{1}{3}) + \frac{1}{4} + \frac{1}{5} = 67/60$ . In either case,  $w_A(B^*) \leq \frac{6}{5}$ .

Suppose  $B^*$  contains a  $Y_3$  item,  $x$ . Since  $w_F(x) \leq \frac{4}{15}$ ,  $w_F(B^*) \leq \frac{1}{3} + \frac{4}{15} + 2(\frac{1}{4}) = 11/10$ . Thus  $w_B(B^*)$  must be more than  $13/10$ , or else  $w_A(B^*)$  cannot exceed  $\frac{6}{5}$ . This implies that  $w_B(x)$  must be  $\frac{1}{3}$ . In this event, there must be a three-item bin with three  $Y_3$  items each of weight  $\frac{1}{3}$ . If there is a  $Y_2$  item of weight  $\frac{1}{2}$ , it must come from an earlier bin containing exactly two  $Y_2$  items. The second of these items would have been replaced, however, by any two of the  $Y_3$  items, since any  $Y_2$  item will fit with any two  $Y_3$  items. Thus there can be no  $Y_2$  items of weight  $\frac{1}{2}$  in the B2F packing, and the weight of the  $Y_2$  item must be  $\frac{2}{5}$ . Therefore,  $w_B(B^*) \leq \frac{2}{5} + \frac{1}{3} + 2(\frac{1}{4}) < 13/10$ .

Suppose now that  $|B^*| = 3$ . If there is no  $X_2$  item or if there is an item of size less than  $\frac{1}{6} + \Delta$ , then  $w_F(B^*) \leq 1$  and  $w_A(B^*) < \frac{6}{5}$ . Thus we may assume that  $B^*$  contains an  $X_2$  item and that its remaining item is at least  $\frac{1}{6} + \Delta$  in size. Even if the small item,  $y$ , is of type  $X_3$ , then  $w_F(B^*)$  is at most  $\frac{1}{2} + 2(\frac{1}{3}) = \frac{7}{6}$ . If the  $Y_2$  item,  $x$ , has B2F weight less than  $\frac{1}{2}$ ,  $w_B(B^*) \leq \frac{1}{2} + \frac{2}{5} + \frac{1}{3}$  and  $w_A(B^*) \leq \frac{6}{5}$ . The only way that  $x$  can have weight  $\frac{1}{2}$  is to be in a two-item bin,  $B_j$ , with another  $Y_2$  item. This means that  $y$  must not have been available when  $B_j$  was packed, since it would have fit with  $x$  and any  $Y_2$  item (it fits in  $B^*$  with  $x$  and an  $X_2$  item). Thus  $y$  cannot have weight  $\frac{1}{3}$  unless there is a three-item bin consisting of items no larger than  $y$ . These items, however, must have been available when  $x$  was packed, and thus  $y$  still cannot have weight  $\frac{1}{3}$ . Therefore, the maximum B2F weight for  $y$  is  $\frac{3}{10}$ .

If  $y$  is a  $Y_3$  item, then  $w_F(y) = \frac{4}{15}$  and  $w_F(B^*) \leq \frac{1}{2} + \frac{1}{3} + \frac{4}{15} = 11/10$ . Thus  $w_B(B^*)$  is at most  $\frac{1}{2} + \frac{1}{2} + \frac{3}{10} = 13/10$  and  $w_A(B^*) \leq \frac{6}{5}$ .

Therefore,  $y$  must be an  $X_3$  item. It must also be that  $\Delta$  exceeds  $\frac{1}{30}$ , or else  $s(B^*) > \frac{5}{12} - \Delta/2 + \frac{1}{3} + \frac{5}{18} - \Delta/3 \geq 1$ . Let  $B_i$  be the bin containing the  $Y_2$  item,  $x$ , in the FFD packing. We know from Lemma 3.2 that  $B_i$  is a  $Y_1$  bin. Let  $z$  be the  $X_2$  item in  $B^*$ . If  $z$  were packed in a  $Y_1$  bin in the FFD packing, its weight would be  $\frac{2}{3}$  and  $w_A(B^*)$  would be  $\leq \frac{6}{5}$ . Thus  $z$  must have been available when  $B_i$  was packed. Since it was not used in place of  $x$ , the size of the  $Y_1$  item in  $B_i$  exceeds  $1 - s(z)$ . But  $z$  fits with  $x$  and  $y$ , so  $s(z) < 1 - \frac{1}{3} - (\frac{5}{18} - \Delta/3) = \frac{7}{18} + \Delta/3$ . Then  $1 - s(z) = 11/18 - \Delta/3$ , which is greater than  $\frac{2}{3} - 2\Delta$  if  $\Delta > \frac{1}{30}$ . The weighting function for FFD gives weight at most  $\frac{4}{15}$  to a  $Y_2$  item packed with such a large  $Y_1$  item, and again we have  $w_A(B^*) \leq \frac{6}{5}$ .

*Case 3.* Suppose  $w_B(B^*) > \frac{6}{5}$  and  $B^*$  contains an item  $a$ , where  $s(a) < \frac{1}{6} + \Delta$ .

We know that  $B^*$  contains neither a  $Y_1$  item nor a  $Y_2$  item of weight exceeding  $\frac{1}{3}$  by Lemma 4.3 and Case 2 above, respectively. We also know that  $w_B(a)$  is at most  $\frac{1}{4}$  since  $s(a) < \frac{1}{4}$ . By Lemma 3.3, we know further that  $w_F(B^*) \leq 1$ , so that if  $w_A(B^*)$  is to exceed  $\frac{6}{5}$ , we must have  $w_B(B^*) > \frac{7}{5}$ . Thus  $|B^*| > 3$ , since any two items with  $a$  can each have weight at most  $\frac{1}{2}$ .

Suppose  $|B^*| = 4$ . Then there must be an  $X_2$  item, or else  $w_B(B^*) \leq 3(\frac{1}{3}) + \frac{1}{4}$ . There can be at most one additional item exceeding  $\frac{1}{4}$  in size, or else  $s(B^*) > \frac{5}{12} - \Delta/2 + 2(\frac{1}{4}) + \frac{1}{6} > 1$ . But then  $w_B(B^*) \leq \frac{1}{2} + \frac{1}{3} + 2(\frac{1}{4}) < \frac{7}{5}$ .

Suppose  $|B^*| = 5$ . There cannot be an  $X_2$  item, or else  $s(B^*) > \frac{5}{12} - \Delta/2 + 4(\frac{1}{6}) > 1$ . Nor can there be two items of size greater than  $\frac{1}{4}$ , or else  $s(B^*) > 2(\frac{1}{4}) + 3(\frac{1}{6}) = 1$ . Finally, if only one item has size exceeding  $\frac{1}{4}$ ,  $w_B(B^*) \leq \frac{1}{3} + 4(\frac{1}{4}) < \frac{7}{5}$ .  $\square$

**THEOREM 5.1.**  $\min \{ \text{FFD}(L), \text{B2F}(L) \} \leq (\frac{6}{5}) \text{OPT}(L) + 8$ .

*Proof.* To obtain this inequality, we observe that our presumed counterexample obeys  $\min \{ \text{FFD}(L), \text{B2F}(L) \} - 8 \leq (\text{FFD}(L) - 8 + \text{B2F}(L) - 8)/2 \leq (w_F(L) + w_B(L))/2 = w_A(L)$  by our definitions for  $w_F$ ,  $w_B$ , and  $w_A$ , while  $w_A(L) \leq (\frac{6}{5}) \text{OPT}(L)$  by Lemma 5.1.  $\square$

**6. Remarks.** We have limited our analysis to proving that, for any list, either the FFD or the B2F algorithm will asymptotically use within  $\frac{6}{5}$  the optimal number of bins. However, we have been unable to find examples that are even close to this bound. In fact, the only examples we have been able to contrive that exceed  $\frac{9}{8}$  the optimum depend heavily on the modification that we introduced to B2F to simplify our proof. For these instances, this modification forces the B2F packing to be the same as the FFD packing. If "small" items are not held back, the exact bound might be significantly better (although a proof of this may well be extremely difficult).

Our weighting function averaging technique actually proves that, even if both algorithms produce particularly egregious packings for some list, the average of the number of bins used by FFD and the number used by B2F is asymptotically at most  $\frac{6}{5}$  the optimal number of bins for that list. Presumably, the minimum may always be considerably less than this upper bound on the average. Furthermore, we remark that the additive constant we have used (eight) is much higher than necessary. Instead of assigning a weight of zero to every exceptional item, we could assign a weight that agrees with an item's type, and easily reduce this constant. Nevertheless, because we believe that the  $\frac{6}{5}$  coefficient is itself inflated, the additive constant appears to be of little significance.

**Appendix. Bin packing results for B2F alone.** We seek to determine the worst-case behavior of the B2F algorithm. Before doing so, however, we briefly discuss some other aspects of this approach to bin packing.

We could extend the idea of "best 2 fit" to "best  $j$  fit," for arbitrary  $j > 2$ . It seems likely that the expected performance of these more complex algorithms might be better,

although the worst-case performance can be shown to be worse, approaching a number greater than 1.3 as  $j$  grows without bound. Simple tests using a uniform distribution for item sizes seem to back up the improved expected case, although the run time increases rapidly.

B2F can also be used in the multifit approach to multiprocessor scheduling. Again, its worst-case performance is poorer than that of FFD. In [3], it is shown that B2F's asymptotic worst-case bound is precisely  $\frac{6}{5}$ , while it has been proved in [4] that FFD can be implemented to ensure a tight bound of  $72/61$ .

Returning to bin packing, Fig. 3 depicts an example illustrating that B2F may require, asymptotically, as many as  $\frac{5}{4}$  the optimal number of bins.

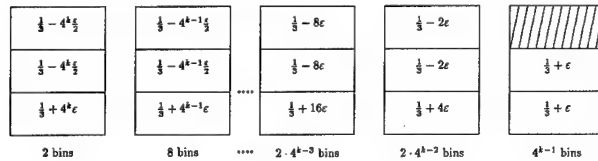
To prove that the  $\frac{5}{4}$  ratio cannot be exceeded by B2F, we modify the algorithm slightly in that items less than or equal to  $\frac{1}{5}$  the bin size will be held back and packed by the FFD algorithm. This certainly does not affect the example illustrated in Fig. 3, but it allows us to assume that no items of size  $\frac{1}{5}$  or less are used in packing  $L$ , which we now presume to be minimal counterexample. This reduces the number of cases we must investigate, thereby simplifying our proof (although it probably detracts from the expected performance of the algorithm).

LEMMA A. *Every item in  $L$  has less than  $\frac{3}{5}$ .*

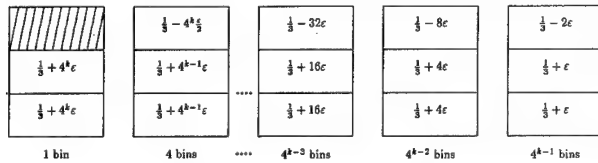
*Proof.* Let  $b$  be the largest item in  $L$  and suppose  $s(b) \geq \frac{3}{5}$ . Then  $b$  is packed in  $B_1$  by the B2F rule. Removing the items of  $B_1$  cannot change the remainder of the packing. Since  $s(b) \geq \frac{3}{5}$ ,  $|B_1| \leq 2$  and, if  $|B_1| = 2$ , then  $B_1$  contains the largest item that would fit with  $b$  in a bin of size 1. If the item or items of  $B_1$  are removed from  $L$ , then both B2F( $L$ ) and OPT( $L$ ) can easily be reduced by one, contradicting the presumed minimality of  $L$  with respect to B2F.  $\square$

THEOREM A.  $\text{B2F}(L) \leq (\frac{5}{4}) \text{OPT}(L) + 4$ .

*Proof.* We classify an item,  $x$ , by its size so that if  $1/(i+1) < s(x) \leq 1/i$ , then  $x$  is of type  $X_i$ . The reasoning above shows that all items are of types  $X_1, X_2, X_3$ , or  $X_4$ , and items of type  $X_1$  are less than  $\frac{3}{5}$  in size. We now define a weighting function  $w$  on the items of  $L$  based on the B2F packing.



$$(a) \quad \text{B2F}(L) = 2(1 + 4 + \dots + 4^{k-2}) + 4^{k-1} = \frac{2(4^{k-1} - 1)}{3} + 4^{k-1} = \left(\frac{5}{3}\right) 4^{k-1} - \frac{2}{3}$$



$$(b) \quad \text{OPT}(L) = 1 + 4 + \dots + 4^{k-2} + 4^{k-1} = \frac{(4^{k-1} - 1)}{3} + 4^{k-1} = \left(\frac{4}{3}\right) 4^{k-1} - \frac{1}{3}$$

FIG. 3. Worst-case example for B2F.  $(\text{B2F}(L)/\text{OPT}(L)) = (5(4^{k-1}) - 2)/(4(4^{k-1}) - 1) \rightarrow 5/4$  as  $k \rightarrow \infty$ .

If  $B$  is any bin with four items in it, each item is assigned a weight of  $\frac{1}{5}$ . Suppose  $B$  is a bin containing an  $X_1$  item,  $b$ . Then if  $|B| = 3$ ,  $w(b) = \frac{8}{15}$ , and the other two items are each assigned a weight of  $\frac{2}{15}$ . If  $|B| = 2$ , then  $w(b) = \frac{8}{15}$  and the other item is assigned a weight of  $\frac{4}{15}$  if the other item is of type  $X_2$ . Otherwise,  $w(b) = \frac{3}{5}$  and the remaining item is assigned a weight of  $\frac{1}{5}$ .

Suppose the largest item in  $B$  is of type  $X_2$ . Then if  $|B| = 2$ , each item must be of type  $X_2$ , and is assigned a weight of  $\frac{2}{5}$ , except possibly for the last bin containing an  $X_2$  item. If the last bin containing an  $X_2$  item has only 2 items in it, it will be classified as exceptional (as will its items). All exceptional items are given weight zero. (This is an unnecessarily strict weight reduction, accounting for the constant 4 in the theorem. A more careful analysis using larger weights for the exceptional items could likely reduce this constant to 1.) If  $|B| = 3$  and  $B$  contains two  $X_2$  items, each is given a weight of  $\frac{3}{10}$  and the remaining item is given a weight of  $\frac{1}{5}$ . If  $B$  contains only one  $X_2$  item, then it is given a weight of  $\frac{2}{5}$  and the other two are each given a weight of  $\frac{1}{5}$ . If the largest item is of type  $X_3$ , then  $|B| = 3$  implies all three items are of type  $X_3$ , except possibly for the last such bin (which is also classified as exceptional). All three  $X_3$  items in such a bin are given a weight of  $\frac{4}{15}$ . One additional exceptional bin shall be identified. If the last  $X_2$  item of size exceeding  $\frac{7}{15}$  is packed with an  $X_2$  item of size less than  $\frac{7}{20}$ , then this bin is classified as exceptional, and its items assigned weights of zero.

The definition of  $w$  is summarized in Table 6.

We now show that each bin  $B^*$  of the optimal packing must satisfy  $w(B^*) \leq 1$ . This, together with the observation that  $w(B) = \frac{4}{5}$  for each nonexceptional bin in the B2F packing, will complete the proof of Theorem A.

Suppose  $B^*$  is a bin of the optimal packing with  $w(B^*) > 1$ . Clearly,  $|B^*| > 1$ . (If  $B^*$  contains an exceptional item, then after removing the item  $w(B^*)$  would still exceed 1. Thus it is enough to show that  $w(B^*) \leq 1$  for bins not containing exceptional items.)

Case 1. Suppose  $|B^*| = 2$ .

If neither item has weight greater than  $\frac{2}{5}$ , then  $w(B^*) \leq \frac{4}{5} < 1$ . Thus  $B^*$  must contain an item of type  $X_1$ . The weight of this item is less than or equal to  $\frac{3}{5}$  and the weight of an  $X_2$  item is less than or equal to  $\frac{2}{5}$ . Since  $B^*$  cannot contain two  $X_1$  items,  $w(B^*) \leq \frac{3}{5} + \frac{2}{5} = 1$ .

Case 2. Suppose  $|B^*| = 3$ .

The largest item in  $B^*$  must have a weight exceeding  $\frac{1}{3}$ , and so must be of type  $X_1$  or  $X_2$ .

TABLE 6  
Weighting function  $w$  used in analysis of B2F alone.

Nonexceptional bin contents	Weights assigned	
$X_1, X_i, X_j$	$\frac{8}{15}, \frac{2}{15}, \frac{2}{15}$	$i, j > 2$
$X_1, X_2$	$\frac{8}{15}, \frac{4}{15}$	
$X_1, X_i$	$\frac{3}{5}, \frac{1}{5}$	$i > 2$
$X_2, X_2$	$\frac{2}{5}, \frac{2}{5}$	
$X_2, X_2, X_i$	$\frac{3}{10}, \frac{3}{10}, \frac{1}{5}$	$i > 2$
$X_2, X_i, X_j$	$\frac{2}{5}, \frac{1}{5}, \frac{1}{5}$	$i, j > 2$
$X_3, X_3, X_3$	$\frac{4}{15}, \frac{4}{15}, \frac{4}{15}$	
any four items	$\frac{1}{5}, \frac{1}{5}, \frac{1}{5}, \frac{1}{5}$	

Suppose the largest item is of type  $X_1$ , so that  $B^* = \{b, c, d\}$ , where  $s(b) > s(c) \geq s(d)$ . Then neither  $c$  nor  $d$  can be of type  $X_2$ . If both are of type  $X_4$ , then  $w(B^*) \leq 1$ . Both cannot be of type  $X_3$ , since  $s(b) + s(c) + s(d)$  cannot exceed 1. Thus  $c$  is of type  $X_3$ ,  $w(c) \leq \frac{4}{15}$ ,  $d$  is of type  $X_4$ , and  $w(d) \leq \frac{1}{5}$ . If both  $c$  and  $d$  were available when  $b$  was packed, then either  $b$  was packed with an  $X_2$  item or with two other items. In either case,  $w(b) = \frac{8}{15}$  and  $w(B^*) \leq \frac{8}{15} + \frac{4}{15} + \frac{1}{5} = 1$ . Therefore,  $w(b)$  must be  $\frac{3}{5}$ . If  $c$  is packed before  $b$ , then  $w(c) \leq \frac{1}{5}$  and  $w(B^*) \leq 1$ . If  $d$  is packed before  $b$ , then it must be packed with an  $X_1$  item and another item, since  $c$  would have fit and was not used. Hence  $w(b) = \frac{2}{5}$  and  $w(B^*) \leq \frac{3}{5} + \frac{4}{15} + \frac{1}{5} = 1$ .

Thus the largest item in  $B^*$  must be of type  $X_2$ . If there is only one  $X_2$  item,  $w(B^*) \leq \frac{2}{5} + 2(\frac{4}{15}) < 1$ . Thus  $B^* = \{b, c, d\}$  with  $b$  and  $c$  both of type  $X_2$ , where  $s(b) \geq s(c)$ . If  $w(d) \leq \frac{1}{5}$ , then  $w(B^*) \leq 2(\frac{2}{5}) + \frac{1}{5} = 1$ . Thus  $d$  is an  $X_3$  item and  $w(d) = \frac{4}{15}$ . Also,  $w(b) = w(c) = \frac{2}{5}$ , since otherwise  $w(B^*) \leq \frac{2}{5} + \frac{3}{10} + \frac{4}{15} < 1$ . Since  $s(d) > \frac{1}{4}$ , it must be that  $s(c) < \frac{3}{8}$ . If  $d$  were packed before  $c$ , then  $w(d)$  would only be  $\frac{1}{5}$ , so that  $d$  must be available. In order for  $w(c)$  to be  $\frac{2}{5}$ ,  $c$  must be packed by B2F in a bin  $B = \{c, x\}$  or  $\{c, y, z\}$ . If  $|B| = 2$ , then since  $d$  would not fit in  $B$ ,  $s(x) > s(b)$  and  $b$  must be in a bin with an  $X_2$  item and one other item, contradicting  $w(b) = \frac{2}{5}$ . If  $|B| = 3$ , then neither  $y$  nor  $z$  can be of type  $X_2$ . Since there must be an  $X_2$  item,  $u$ , left (or else  $B$  would be exceptional) and since  $u$  is smaller than  $c$ , the B2F rule would have placed  $c, u$ , and an  $X_3$  item in  $B$  since  $c, u$ , and  $d$  would have fit. Thus it is impossible to have  $w(b) = w(c) = \frac{2}{5}$  while  $w(d) = \frac{4}{15}$  and we conclude that, in any event,  $w(B^*) \leq 1$ .

Case 3. Suppose  $|B^*| = 4$ .

$B^*$  cannot contain an  $X_1$  item, since  $\frac{1}{2} + 3(\frac{1}{5}) > 1$ . Neither can it contain two  $X_2$  items, since  $2(\frac{1}{3}) + 2(\frac{1}{5}) > 1$ . Similarly, it cannot contain four  $X_3$  items, since each has size greater than  $\frac{1}{4}$ . However, if it contains three items of type  $X_3$  and one of type  $X_4$ , then  $w(B^*) \leq 3(\frac{4}{15}) + \frac{1}{5} = 1$ . Thus  $B^*$  must contain exactly one  $X_2$  item. If the other three items have weight less than or equal to  $\frac{1}{5}$ ,  $w(B^*) \leq \frac{2}{5} + 3(\frac{1}{5}) = 1$ . If there were two  $X_3$  items,  $s(B^*) > \frac{1}{3} + 2(\frac{1}{4}) + \frac{1}{5} > 1$ . Thus  $B^*$  must contain exactly one  $X_3$  item. Let  $B^* = \{b, c, d, e\}$ , with  $b$  of type  $X_2$ , and  $c$  of type  $X_3$ . If  $w(b) < \frac{2}{5}$ , then  $w(B^*) \leq \frac{3}{10} + \frac{4}{15} + 2(\frac{1}{5}) < 1$ . Thus  $w(b) = \frac{2}{5}$  and  $w(c) = \frac{4}{15}$ . This means  $c$  must be available when  $b$  is packed.

If  $b$  is the largest item in some bin  $B$  of the B2F packing, then  $B$  would contain two  $X_2$  items and another item since  $s(b) + s(c) + s(d) + s(e) \leq 1$  implies that  $2s(b) + s(c) < 1$ . This cannot happen, however, so it must be that  $B = \{x, b\}$  where  $s(x) > 1 - 2s(c)$ , since  $b$  was not replaced by two smaller items. Because  $s(c) < 1 - \frac{1}{3} - \frac{2}{5} = \frac{4}{15}$ , we know  $s(x) > \frac{7}{15}$ . Thus  $B$  is the third exceptional bin ( $s(b) < 1 - \frac{1}{4} - \frac{2}{5} = \frac{7}{20}$ ) and again  $w(B^*) \leq 1$ .

Now, to complete our proof of Theorem A, we note that  $w(B) = \frac{4}{5}$  for all but at most four B2F bins (the three exceptional bins and the last bin), so that  $\sum_{x \in L} w(x) \geq (\frac{4}{5})(\text{B2F}(L) - 4)$ . At the same time,  $w(B^*) \leq 1$  for all  $B^*$  in the optimal packing ensures  $\sum_{x \in L} w(x) \leq \text{OPT}(L)$ . Combining these two inequalities yields  $\text{B2F}(L) \leq (\frac{5}{4})\text{OPT}(L) + 4$ , as desired.  $\square$

#### REFERENCES

- [1] W. FERNANDEZ DE LA VEGA AND G. S. LUEKER, *Bin packing can be solved within  $1 + \epsilon$  in linear time*, *Combinatorica*, 1 (1981), pp. 349–355.

- [2] G. N. FREDERICKSON, M. S. HECHT, AND C. E. KIM, *Approximation algorithms for some routing problems*, SIAM J. Computing, 7 (1978), pp. 178–193.
- [3] D. K. FRIESEN, *Analysis of a new bin packing algorithm and its application to scheduling*, Computer Science Technical Report, Texas A&M University, College Station, Texas.
- [4] D. K. FRIESEN AND M. A. LANGSTON, *Evaluation of a multifit-based scheduling algorithm*, J. Algorithms, 7 (1986), pp. 35–59.
- [5] M. R. GAREY AND D. S. JOHNSON, *A  $7\frac{1}{2}/60$  theorem for bin packing*, J. Complexity, 1 (1985), pp. 65–106.
- [6] D. S. JOHNSON, *Near optimal bin packing algorithms*, Ph.D. thesis, M.I.T. Cambridge, MA, 1973.
- [7] N. KARMARKAR AND R. M. KARP, *An efficient approximation scheme for the one-dimensional bin packing problem*, Proc. 23rd Symp. on Foundations of Computer Science, Chicago, IL, 1982, pp. 312–320.
- [8] M. A. LANGSTON, *Interstage transportation planning in the deterministic flowshop environment*, Oper. Res., 35 (1987), pp. 556–564.
- [9] A. C. YAO, *New algorithms for bin packing*, J. ACM, 27 (1980), pp. 207–227.

## SEMIKERNELS, QUASI KERNELS, AND GRUNDY FUNCTIONS IN THE LINE DIGRAPH\*

H. GALEANA-SÁNCHEZ†, L. PASTRANA RAMÍREZ‡, AND H. A. RINCÓN-MEJÍA‡

**Abstract.** It is proved that the number of semikernels (quasi kernels) of a digraph  $D$  is less than or equal to the number of semikernels (quasi kernels) of its line digraph  $L(D)$ . It is also proved that the number of Grundy functions of  $D$  is equal to the number of Grundy functions of its line digraph  $L(D)$  (in the case where every vertex of  $D$  has indegree at least one).

**Key words.** Grundy function, kernel, line digraph, quasi kernel, semikernel

**AMS(MOS) subject classification.** 05C20

**1. Introduction.** For general concepts we refer the reader to [1]. Let  $D = (X, U)$  be a digraph (also we denote  $X = V(D)$  and  $U = A(D)$ ). A set  $K \subseteq X$  is said to be a kernel if it is both independent (a vertex in  $K$  has no successor in  $K$ ) and absorbing (a vertex not in  $K$  has a successor in  $K$ ).

This concept was introduced by Von Neumann [10] and it has found many applications [1, p. 304], [2]. Several authors have been investigating sufficient conditions for the existence of kernels in digraphs, namely, Von Neumann and Morgenstern [9], Richardson [11], Duchet and Meyniel [4], [5], and Galeana-Sánchez and Neumann-Lara [7].

In [8] Harminec proved that the number of kernels of a digraph is equal to the number of kernels in its line digraph. In this paper we find similar relations for concepts nearly related to the concept of kernel, and we survey the theorems relating these concepts.

**DEFINITION 1.1** [10]. A semikernel  $S$  of  $D$  is an independent set of vertices such that for every  $z \in (V(D) - S)$  for which there exists a  $Sz$ -arc there also exists an  $zS$ -arc.

**DEFINITION 1.2** [3]. A quasi kernel  $Q$  of  $D$  is an independent set of vertices such that  $X = Q \cup \Gamma^-(Q) \cup \Gamma^-(\Gamma^-(Q))$  (where for any  $A \subseteq X$ ,  $\Gamma^-(A) = \{x \in X \mid x \text{ has a successor in } A\}$ ).

**DEFINITION 1.3** [1, p. 312]. A nonnegative integer function  $g(x)$  is called a Grundy function of  $D$  if, for every vertex  $x$ ,  $g(x)$  is the smallest nonnegative integer which does not belong to the set  $\{g(y) \mid y \in \Gamma^+(x)\}$ .

This concept, originated by Grundy for digraphs without directed cycles, was extended by Berge and Schützenberger.

The Grundy function can also be defined as a function  $g(x)$  such that

(1)  $g(x) = k > 0$  implies that for each  $0 \leq j < k$  there is a  $y \in \Gamma^+(x)$  with  $g(y) = j$ .

(2)  $g(x) = k$  implies that each  $y \in \Gamma^+(x)$  satisfies  $g(y) \neq k$ .

**THEOREM 1.1** [3]. Every finite digraph has a quasi kernel. A generalization of this theorem was obtained by Duchet, Hamidoune, and Meyniel [6].

**THEOREM 1.2** [10]. If  $D$  is a digraph such that every induced subdigraph has a nonempty semikernel then  $D$  has a kernel.

**THEOREM 1.3** [1, p. 313]. If  $D$  is a digraph such that every induced subdigraph has a kernel then  $D$  possesses a Grundy function.

\* Received by the editors April 5, 1989; accepted for publication December 18, 1989.

† Instituto de Matemáticas de la Universidad Nacional Autónoma de México, México, D.F.

‡ Departamento de Matemáticas de la Facultad de Ciencias de la Universidad Nacional Autónoma de México, México, D.F.

## Stable Duplicate-Key Extraction with Optimal Time and Space Bounds<sup>\*</sup>

Bing-Chao Huang<sup>1,\*\*\*</sup> and Michael A. Langston<sup>2,\*\*\*</sup>

<sup>1</sup> Department of Computer Science, University of South Carolina, Columbia, SC 29208, USA

<sup>2</sup> Department of Computer Science, Washington State University, Pullman, WA 99164-1210, USA

**Summary.** We consider the problem of transforming a list  $L$  of records sorted on some key into two sublists  $L_1$  and  $L_2$  where, for each distinct key in  $L$ ,  $L_1$  contains the first record of  $L$  that possesses the key and  $L_2$  contains all records of  $L$  with duplicate keys. We desire that our duplicate-key extraction algorithm perform the transformation in place and be stable (that is, records within each sublist must obey the original order given by  $L$ ). This operation is useful in database and related file processing environments whenever only distinct keys need be considered. Moreover, stability in extraction insures that  $L$  can be efficiently restored at a later time with a stable merge of  $L_1$  and  $L_2$ . Any procedure for performing duplicate-key extraction on a list of size  $n$  must require at least  $O(n)$  time and  $O(1)$  extra space, although the obvious algorithm for achieving either bound alone violates the other bound. We design a stable algorithm, using block-rearrangement techniques, and show that it is optimal in the theoretical sense that it achieves both lower bounds simultaneously. We also prove that its worst-case number of key comparisons and record exchanges sum to no more than  $6n$ , suggesting that the algorithm has practical application as well.

### 1. Introduction

Suppose that we are given a list  $L$  with  $n$  records sorted on some key, and that some records may possess the same key. A variety of questions about such lists focus only on distinct keys. For example, one might want to know whether a list contains a particular key, whether two or more lists contain the same keys, whether all keys are present within a certain range, etc. Repeated queries of this nature frequently arise in database and file processing environments, and can often be most efficiently addressed by first extracting duplicate keys from  $L$ , leaving only one copy of each distinct key.

<sup>\*</sup> A preliminary version of a portion of this paper [HL1] was presented at the 24th Annual Allerton Conference on Communication, Control and Computing held in Monticello, Illinois, in October, 1986

<sup>\*\*</sup> This author's research has been supported in part by the Washington State University Graduate Research Assistantship Program

<sup>\*\*\*</sup> This author's research has been supported in part by the National Science Foundation under grants ECS-8403859 and MIP-8603879, and by the Office of Naval Research under contract N00014-88-K-0343

If  $n$  is large, then using an additional list to hold a copy of each of  $L$ 's distinct keys may require a prohibitive amount of additional storage. Therefore, we focus our attention on schemes that extract duplicates *in-place*, so that  $L$  is transformed into the concatenation of two sublists,  $L_1$  and  $L_2$ , where  $L_1$  contains one record for each distinct key in  $L$  and  $L_2$  contains  $L - L_1$ . We allow the use of a few extra memory cells to aid in the transformation, but their total number must be constant. In-place extraction, however, dictates that the operation be invertible (since  $L$  is altered), so that we can restore  $L$  to its original sequence if necessary. Hence we ask that  $L_1$  contain the *first* copy of each distinct key and that our algorithm be *stable*, by which we mean that keys within each sublist retain the relative order they held within  $L$ . Clearly a subsequent in-place, stable merge of  $L_1$  and  $L_2$  restores  $L$  (see, for example, [Tr, SS2, HL4] for increasingly-efficient, asymptotically-optimal versions of such a merge).

Any algorithm to solve this problem needs  $\Omega(n)$  time since, in general, every key must be examined at least once to determine whether it is duplicated within  $L$ . There is, of course, an obvious way to construct  $L_1$  and  $L_2$  in  $O(n)$  time, by a linear scan of  $L$  and temporary use of a list of storage cells separate from  $L$ . This simple method is, unfortunately, unacceptable since  $\Omega(n)$  additional space must be available. Similarly, there is a straightforward way to construct  $L_1$  and  $L_2$  in-place in  $O(1)$  extra space, by finding each new element of  $L_1$  in turn and moving it to its final position by shifting duplicates out of the way. In this case, however,  $\Omega(n^2)$  time may be required just to move records. In the sequel, we will show that a more complicated strategy, which uses  $O(\sqrt{n})$  blocks of size  $O(\sqrt{n})$ , solves the problem in both  $O(n)$  time and  $O(1)$  space, and hence is optimal to within a constant factor.

In the following section, we discuss previous and ongoing work as it relates to the main results of this paper, as well as to this general topic. In Sect. 3, we introduce some necessary notation and define a few useful, primitive suboperations. Section 4 contains an overview of the main algorithm along with an example, where simplifying assumptions are made on list, sublist and block sizes in order to facilitate discussion. Time and space measures are derived. In Sect. 5, we describe minor implementation details that permit the simplifying assumptions to be dropped. We also include a listing of our algorithm furnished to us by the editor, Michael J. Fischer, who was kind enough to take the interest to implement our technique in Pascal. Section 6 contains an analysis that shows that the worst-case number of key comparisons and record exchanges sum to no more than  $6n$ , suggesting the algorithm may be of practical as well as theoretical interest. In the final section, we present a brief discussion of some conclusions that can be drawn from this effort.

## 2. Related Work

The efficiency we shall achieve in breaking  $L$  into  $O(\sqrt{n})$  blocks, each of size  $O(\sqrt{n})$ , inherently relies on the notions of *internal buffering* and *block rearranging*, which can be traced back to the seminal work described in [Kr]. As applied

in this paper, this general approach allows us to employ one block as the buffer to aid in grouping the remainder of  $L$  into blocks of different types. Since only the contents of the buffer and the relative order of the blocks need end up out of sequence, linear time is sufficient to complete our task by sorting both the buffer and the blocks (each sort involves  $O(\sqrt{n})$  keys).

Previously published results have only shown merging (and hence sorting by merging) to yield to this line of attack. An unstable method was first devised in [Kr]. A stable scheme was later proposed in [Ho] that, unfortunately, had the rather undesirable side-effect that records had to be alterable during its execution. Subsequently, a general algorithm for optimal time and space, stable merging and sorting appeared in [Tr]. For the most part, however, these results have been of theoretical interest only, due primarily to their intricacy and prohibitively large time complexity constants of proportionality.

Continued research efforts have focused on simpler, more practical internal buffering and block rearranging strategies for optimal time and space unstable merging [HL2, MU] and stable merging [HL4, SS2], as well as even faster stable sorting schemes [HL4] that bypass the obvious merge-sort implementation. It has also been shown that unmerging is amenable to this general approach [SS1], although information other than a record's key alone must be available. Furthermore, we have very recently found [HL3] that all of the elementary binary set and multiset operations can be performed on sorted lists in linear time and constant extra space, with potential application to a number of file processing problems.

The primary aim of this paper is to demonstrate that stable duplicate-key extraction is possible in optimal time and space. We shall also show that, relative to previously published results along this line, our algorithm is straightforward and practical. Important differences between the techniques used in the well-known merge strategy of [Tr] and the methods we employ in the duplicate-key extraction method we present herein include these: 1) we pass the buffer directly across the list so as to minimize unnecessary record movement, 2) we avoid tedious complications in the special case in which there are not enough distinct keys to fill the buffer, and 3) we delay buffer resequencing until the final step, thereby sorting it only once.

### 3. Notation and Preliminaries

Let  $L$  denote a list of  $n$  records, indexed from 1 to  $n$ . We use  $KEY(i)$  as a shorthand to denote the key of the record with index  $i$ , and assume that  $L$  is sorted in nondecreasing order so that  $KEY(1) \leq KEY(2) \leq \dots \leq KEY(n)$ . For the sake of complete generality, we allow neither the key nor any other part of a record to be modified during duplicate-key extraction. Such is necessary, for example, when there is no explicit key field within each record, but instead a record's key is a function of one or more of its data fields. We use the term  $L_1$  record to denote one that is to go to  $L_1$ . That is, an  $L_1$  record either has index 1 or it has index  $i$ , for some  $1 < i \leq n$ , where  $KEY(i-1) < KEY(i)$ . We employ the term  $L_2$  record in an analogous fashion to denote one that is to go to  $L_2$ .

Only the two common  $O(1)$  time and space elementary operations are assumed, namely, record exchanges and key comparisons. The exchange procedure,  $SWAP(i, j)$ , directs that the  $i$ th and  $j$ th records are to be exchanged. The comparison functions, for example  $KEY(i) < KEY(j)$ , return the expected Boolean values dependent on the relative values of the keys being compared.

From these primitive operations we construct a few  $O(1)$  space useful subprograms for dealing with *blocks*. Let us define a block to be a set of records from  $L$  with consecutive indices. The *head* of a block is the record with the lowest index; the *tail* is the one with the highest index. If a block contains only  $L_1(L_2)$  records, then we refer to it as an  $L_1(L_2)$  block. The procedure  $BLOCKSWAP(i, j, h)$  exchanges a block of  $h$  records beginning at index  $i$  with a block of  $h$  records beginning at index  $j$  in  $O(h)$  time. We specify that blocks do not partially overlap (i.e., if  $i \neq j$  then  $h \leq |i - j|$ ) and that when  $BLOCKSWAP$  is finished, records within a moved block retain the order they possessed before  $BLOCKSWAP$  was invoked. A block of  $h$  records beginning at index  $i$  is sorted in nondecreasing order by the procedure  $SORT(i, h)$ . Finally, the procedure  $BLOCKSORT(i, h, p)$  uses  $BLOCKSWAP$  to rearrange the  $p$  consecutive blocks, each with  $h$  records, beginning at index  $i$  so that their heads are sorted in nondecreasing order. To reduce unnecessary record movement, which is important should records be relatively long, we assume  $SORT$  and  $BLOCKSORT$  use a straight selection sort [Kn], yielding respective time complexities  $O(h^2)$  and  $O(p^2 + ph)$ .

#### 4. An Overview of the Main Algorithm

In order to facilitate discussion, let us assume for the moment that  $n$  is a perfect square, with  $a\sqrt{n}L_1$  records and  $b\sqrt{n}L_2$  records, where  $a$  and  $b$  are positive integers and  $a + b = \sqrt{n}$ . Figure 1 depicts such a list with  $a = 3$ ,  $b = 3$  and  $n = 36$ . Only record keys are listed, represented by capital letters. Subscripts are added to keep track of duplicate keys as the algorithm progresses.

The first step of the algorithm is to fill an "internal buffer" of size  $\sqrt{n}$  with the  $\sqrt{n}$  largest-keyed  $L_1$  records (their order within the buffer is not important). Thus we convert  $L$  into the form  $ABC$  where the records of  $A$  have not been disturbed,  $B$  is the buffer, and  $C$  is a suffix (rightmost sublist) of  $L_2$ .  $B$  is constructed by conducting a right-to-left scan of  $L$ . When a comparison of adjacent keys reveals that a new  $L_1$  record has been found, the record is included in  $B$ . When an  $L_2$  record is encountered, it is exchanged with the current rightmost element of  $B$ . Therefore,  $B$  begins with size zero at the right end of  $L$  and grows as we "roll" it toward the left until it accumulates  $\sqrt{n}L_1$  records, now unordered by key. For simplicity, we assume that the size of  $A$ , and hence  $C$ , is an integral multiple of  $\sqrt{n}$ . Figure 1 illustrates how this process modifies our example list of 36 elements.

It should be clear that, at this point, no more than  $n$  key comparisons and  $n$  record exchanges are needed. Also, a couple of additional storage cells are enough to keep track of the buffer's position.

$A_1 A_2 B_1 C_1 C_2 D_1 D_2 D_3 E_1 E_2 E_3 E_4 F_1 F_2 G_1 H_1 I_1 I_2 J_1 K_1 K_2 M_1 M_2 M_3 N_1 O_1 O_2 P_1 Q_1 Q_2 R_1 R_2 R_3 S_1 S_2 S_3$

a) Example list  $L$ , with  $a = 3, b = 3$  and  $n = 36$ .

$A_1 A_2 B_1 C_1 C_2 D_1 D_2 D_3 E_1 E_2 E_3 E_4 F_1 F_2 G_1 H_1 I_1 I_2 J_1 K_1 K_2 M_1 M_2 M_3 N_1 O_1 O_2 P_1 Q_1 Q_2 R_1 R_2 R_3 \underbrace{S_1}_{B} S_2 S_3$

b) First buffer element is found.

$A_1 A_2 B_1 C_1 C_2 D_1 D_2 D_3 E_1 E_2 E_3 E_4 F_1 F_2 G_1 H_1 I_1 I_2 J_1 K_1 K_2 M_1 M_2 M_3 N_1 O_1 O_2 P_1 Q_1 Q_2 \underbrace{R_1 S_1}_{B} R_2 R_3 S_2 S_3$

c) Second buffer element is found.

$\underbrace{A_1 A_2 B_1 C_1 C_2 D_1 D_2 D_3 E_1 E_2 E_3 E_4 F_1 F_2 G_1 H_1 I_1 I_2 J_1 K_1 K_2 M_1 M_2 M_3}_{A} \underbrace{N_1 O_1 R_1 P_1 Q_1 S_1}_{B} \underbrace{O_2 Q_2 R_2 R_3 S_2 S_3}_{C}$

d) Buffer is filled.

Fig. 1. Filling the internal buffer,  $B$

The second step of the algorithm is the most complex. We transform  $ABC$  into  $XYC$ , where  $X$  contains  $a$   $L_1$  blocks (one of which is  $B$ ) and  $YC = L_2$ . This is accomplished as follows.  $B$  is used to partition  $A$  into a collection of  $\sqrt{n}$ -sized  $L_1$  and  $L_2$  blocks.  $B$ 's initial position for this step will become the rightmost  $L_2$  block of  $Y$ . The  $\sqrt{n}$  memory cells to its immediate left will become an  $L_1$  block.  $A$  is scanned from right to left until an  $L_2$  record is found, which is then exchanged with the buffer's tail. The scan is continued, each record in turn being exchanged with the rightmost buffer element in the appropriate block. Thus the buffer is, in general, broken into two pieces, each to the left of the growing edge of a block (see Fig. 2d). When an  $L_1$  block is filled, a new  $L_1$  block is begun in the set of  $\sqrt{n}$  cells to its immediate left. When an  $L_2$  block is filled, *BLOCKSWAP* is invoked if necessary to move it to its final position, ousting an  $L_1$  block there. Observe that handling  $L_2$  blocks in this manner insures that  $Y$  will contain the appropriate prefix (leftmost sublist) of  $L_2$ . Figure 2 shows how such an  $L_2$  block is constructed.

A new  $L_2$  block is begun in the position of the leftmost  $L_1$  block, after its  $c$   $L_1$  records,  $0 \leq c < \sqrt{n}$ , are exchanged with the rightmost  $c$  records, all from  $B$ , in the block to its immediate left. (Therefore a new  $L_2$  block initially contains all of  $B$ . See Fig. 3b.) When all of  $A$  has been scanned in the manner described above, every block of  $X$ , except for  $B$ , remains sorted internally. The blocks themselves may, however, be unordered with respect to each other. Figure 3 depicts how the transformation to  $XYC$  is completed on our example list.

Therefore, during the second step of the algorithm, as the sublist  $AB$  is transformed into  $XY$  it takes on the general form

$$U, B_1, P_1, Q_1, \dots, Q_k, B_2, P_2, Q_{k+1}, \dots, Q_l, V$$

$A_1 A_2 B_1 C_1 C_2 D_1 D_2 D_3 E_1 E_2 E_3 E_4 F_1 F_2 G_1 H_1 I_1 I_2 J_1 K_1 K_2 M_1 M_2 M_3 \underbrace{N_1 O_1 R_1 P_1 Q_1 S_1}_B \underbrace{O_2 Q_2 R_2 R_3 S_2 S_3}_C$

a) Example list after buffer is filled.

$A_1 A_2 B_1 C_1 C_2 D_1 D_2 D_3 E_1 E_2 E_3 E_4 F_1 F_2 G_1 H_1 I_1 I_2 J_1 K_1 K_2 M_1 M_2 \underbrace{S_1 N_1 O_1 R_1 P_1 Q_1}_B M_3 \underbrace{O_2 Q_2 R_2 R_3 S_2 S_3}_C$

b) First  $L_2$  record is moved.

$A_1 A_2 B_1 C_1 C_2 D_1 D_2 D_3 E_1 E_2 E_3 E_4 F_1 F_2 G_1 H_1 I_1 I_2 J_1 K_1 K_2 M_1 \underbrace{Q_1 S_1 N_1 O_1 R_1 P_1}_B M_2 M_3 \underbrace{O_2 Q_2 R_2 R_3 S_2 S_3}_C$

c) Second  $L_2$  record is moved.

$A_1 A_2 B_1 C_1 C_2 D_1 D_2 D_3 E_1 E_2 E_3 E_4 F_1 F_2 G_1 H_1 I_1 I_2 J_1 K_1 K_2 \underbrace{S_1 Q_1}_B M_1 \underbrace{N_1 O_1 R_1 P_1}_B M_2 M_3 \underbrace{O_2 Q_2 R_2 R_3 S_2 S_3}_C$

d) First  $L_1$  record is moved.

$A_1 A_2 B_1 C_1 C_2 D_1 D_2 D_3 E_1 E_2 E_3 \underbrace{N_1 R_1 O_1 S_1 Q_1 P_1}_B F_1 \underbrace{G_1 H_1 I_1 J_1 K_1 M_1}_{\text{an } L_1 \text{ block}} \underbrace{E_4 F_2 I_2 K_2 M_2 M_3}_{\text{an } L_2 \text{ block}} \underbrace{O_2 Q_2 R_2 R_3 S_2 S_3}_C$

e) An  $L_2$  block is completed (in its final position).

Fig. 2. Constructing an  $L_2$  block

where  $B_1 B_2$  is the buffer,  $P_1$  is a partially-filled  $L_1$  block,  $P_2$  is a partially-filled  $L_2$  block, each  $Q_j$  is an  $L_1$  block of size  $\sqrt{n}$ , and  $V$  consists solely of  $L_2$  records. The size of  $B_2 P_2$  is exactly  $\sqrt{n}$ . The basic operation is to swap the last record of  $U$  with the last record of  $B_1$  or  $B_2$ , depending on whether it is an  $L_1$  or an  $L_2$  record. This causes  $U$  and  $B_1$  or  $B_2$  to shrink on the right by one element and the blocks to their immediate right to grow accordingly. If  $P_1$  becomes full (i.e., contains  $\sqrt{n}$  records), it is relabelled as a  $Q$  block and a new empty  $P_1$  block is placed immediately to its left. This basic operation is repeated until  $B_2$  becomes empty, at which point  $P_2$  is a complete  $L_2$  block.  $P_2$  is then swapped with  $Q_l$ , placing it adjacent to  $V$ , and yielding the configuration

$$U, B_1, P_1, Q_1, \dots, Q_k, Q_l, Q_{k+1}, \dots, Q_{l-1}, P_2, V.$$

To continue, the  $Q$  blocks are renumbered, and the old  $P_2 V$  sublist is relabelled as  $V$ . The entire buffer is now in  $B_1$ , but it is not necessarily aligned on a block boundary since, in general,  $P_1$  is shorter than  $\sqrt{n}$ . To align the buffer,  $P_1$  is swapped with the first part of  $B_1$  giving

$$U, P_1, B_2, Q_1, \dots, Q_k, Q_{k+1}, \dots, Q_l, V$$

where  $B_2$ , the new buffer, is a rotation of the old  $B_1$ . The induction continues by placing an empty  $B_1$  in front of  $P_1$  and an empty  $P_2$  behind  $B_2$ , again producing the general form (now with  $k=0$ )

$$U, B_1, P_1, B_2, P_2, Q_1, \dots, Q_l, V.$$

$A_1 A_2 B_1 C_1 C_2 D_1 D_2 D_3 E_1 E_2 E_3 \underbrace{N_1 R_1 O_1 S_1 Q_1 P_1}_{B} F_1 \underbrace{G_1 H_1 I_1 J_1 K_1 M_1}_{\text{an } L_1 \text{ block}} \underbrace{E_4 F_2 I_2 K_2 M_2 M_3}_{\text{an } L_2 \text{ block}} \underbrace{O_2 Q_2 R_2 R_3 S_2 S_3}_{C}$

a) Example list after  $L_2$  block is completed.

$A_1 A_2 B_1 C_1 C_2 D_1 D_2 D_3 E_1 E_2 E_3 F_1 \underbrace{R_1 O_1 S_1 Q_1 P_1 N_1}_{B} \underbrace{G_1 H_1 I_1 J_1 K_1 M_1}_{\text{an } L_1 \text{ block}} \underbrace{E_4 F_2 I_2 K_2 M_2 M_3}_{\text{an } L_2 \text{ block}} \underbrace{O_2 Q_2 R_2 R_3 S_2 S_3}_{C}$

b) An  $L_1$  record is moved so that another  $L_2$  block can be started.

$A_1 \underbrace{R_1 Q_1 N_1 O_1 P_1 S_1}_{B} B_1 C_1 D_1 E_1 F_1 \underbrace{A_2 C_2 D_2 D_3 E_2 E_3}_{\text{an } L_2 \text{ block}} \underbrace{G_1 H_1 I_1 J_1 K_1 M_1}_{\text{an } L_1 \text{ block}} \underbrace{E_4 F_2 I_2 K_2 M_2 M_3}_{\text{an } L_2 \text{ block}} \underbrace{O_2 Q_2 R_2 R_3 S_2 S_3}_{C}$

c) Next  $L_2$  block is completed.

$A_1 \underbrace{R_1 Q_1 N_1 O_1 P_1 S_1}_{B} B_1 C_1 D_1 E_1 F_1 \underbrace{G_1 H_1 I_1 J_1 K_1 M_1}_{\text{an } L_1 \text{ block}} \underbrace{A_2 C_2 D_2 D_3 E_2 E_3}_{\text{an } L_2 \text{ block}} \underbrace{E_4 F_2 I_2 K_2 M_2 M_3}_{\text{an } L_2 \text{ block}} \underbrace{O_2 Q_2 R_2 R_3 S_2 S_3}_{C}$

d)  $L_2$  block is moved to final position.

$\underbrace{A_1 B_1 C_1 D_1 E_1 F_1 R_1 Q_1 N_1 O_1 P_1 S_1}_{\text{an } L_1 \text{ block}} \underbrace{B_1}_{B} \underbrace{G_1 H_1 I_1 J_1 K_1 M_1}_{\text{an } L_1 \text{ block}} \underbrace{A_2 C_2 D_2 D_3 E_2 E_3}_{\text{an } L_2 \text{ block}} \underbrace{E_4 F_2 I_2 K_2 M_2 M_3}_{\text{an } L_2 \text{ block}} \underbrace{O_2 Q_2 R_2 R_3 S_2 S_3}_{C}$   
 $X \qquad \qquad \qquad Y$

e)  $L_1$  records are moved.  $L_2$  sublist is completed.

Fig. 3. Finishing the  $L_2$  sublist

When  $U$  is exhausted,  $P_1$  and  $P_2$  will both be empty under our assumption that the numbers of  $L_1$  and  $L_2$  records are both multiples of  $\sqrt{n}$ .

At most  $n$  comparisons are needed to compare adjacent keys in this second step.  $O(n)$  time suffices for record movement as well, since there are at most  $n$  record exchanges and at most  $2\sqrt{n}$  block exchanges. Only a small, constant number of additional storage cells are needed for counters and pointers.

The final step of the algorithm is to transform  $X$  into  $L_1$ . To do this,  $B$  is first sorted (recall that all keys in  $B$  are distinct, insuring stability). Then *BLOCKSORT* is used to sort the blocks of  $X$  by their heads. Since these blocks were constructed one at a time, and since each is sorted internally,  $L_1 L_2$  is the final result. See Fig. 4.

$O(n)$  time and  $O(1)$  space are sufficient for this final step, since each sort involves at most  $\sqrt{n}$  keys.

## 5. Implementation Details

We now describe the necessary implementation details that permit us to perform duplicate-key extraction without regard to any assumptions about actual list, sublist or block sizes. Let  $s = \lfloor \sqrt{n} \rfloor$  denote the size we will use for a block. Thus the number of  $L_1$  records is  $st_1 + e_1$ , for some unique nonnegative integers  $t_1$  and  $e_1$ , where  $e_1 < s$ . Similarly, the number of  $L_2$  records can be denoted

$\underbrace{A_1 B_1 C_1 D_1 E_1 F_1}_{\text{an } L_1 \text{ block}} \underbrace{R_1 Q_1 N_1 O_1 P_1 S_1}_B \underbrace{G_1 H_1 I_1 J_1 K_1 M_1}_{\text{an } L_1 \text{ block}} \underbrace{A_2 C_2 D_2 D_3 E_2 E_3 E_4 F_2 I_2 K_2 M_2 M_3 O_2 Q_2 R_2 R_3 S_2 S_3}_{L_2}$

a) Example list after  $L_2$  sublist completed.

$\underbrace{A_1 B_1 C_1 D_1 E_1 F_1}_{\text{an } L_1 \text{ block}} \underbrace{N_1 O_1 P_1 Q_1 R_1 S_1}_B \underbrace{G_1 H_1 I_1 J_1 K_1 M_1}_{\text{an } L_1 \text{ block}} \underbrace{A_2 C_2 D_2 D_3 E_2 E_3 E_4 F_2 I_2 K_2 M_2 M_3 O_2 Q_2 R_2 R_3 S_2 S_3}_{L_1}$

b) B is sorted.

$A_1 B_1 C_1 D_1 E_1 F_1 G_1 H_1 I_1 J_1 K_1 M_1 N_1 O_1 P_1 Q_1 R_1 S_1 A_2 C_2 D_2 D_3 E_2 E_3 E_4 F_2 I_2 K_2 M_2 M_3 O_2 Q_2 R_2 R_3 S_2 S_3$

c) BLOCKSORT is performed on blocks of X.  $L_1$  sublist is completed.

Fig. 4. Finishing the  $L_1$  sublist

by the expression  $st_2 + e_2$ ,  $e_2 < s$ . Note that  $(s+1)^2 > n \geq st_1 + st_2$  guarantees that the total number of  $s$ -sized blocks,  $t_1 + t_2$ , is no more than  $s+2$ . We now consider the file as a collection of  $t_1 + t_2 + 2$  blocks, the first a (possibly empty) block of size  $e_1$ , followed by the  $t_1 + t_2$   $s$ -sized blocks, followed by a final (possibly empty) block of size  $e_2$ .

First, we attempt to fill the internal buffer. Observe that in doing so we may have examined all  $L_2$  records, in which case we merely sort the (perhaps only partially filled) buffer and halt. Otherwise, we scan the remainder of  $L$  to determine the number of  $L_1$  records, from which we derive the value of  $e_1$ . The block containing the rightmost buffer element becomes the first  $L_2$  block, the block to its immediate left the first  $L_1$  block. We are thus either finished with the last block, which is of size  $e_2$ , or soon will be. Its records will end up in their final position; its unusual size can cause no problem.

We next initiate the second step of the algorithm as outlined in the previous section. When the last  $L_2$  block has been filled and moved, we know that there is no need to continue scanning  $A$ . Note that, with the final exchange of fewer than  $s$   $L_1$  and buffer records, the buffer now occupies one full block, ensuring that we are finished with the first block (which has the unusual size  $e_1$ ) as well as any other block to the left of the buffer.

We complete the job of duplicate-key extraction by sorting and moving the buffer to its final position, then invoking *BLOCKSORT* on the  $L_1$  blocks that may need rearrangement. Hence the following detailed description of algorithm *EXTRACT* and its two subprograms *BUFFERFILL* and *BLOCKIFY*. Our original (and, we confess, rather abstruse) description in pidgin Algol [HL1] is herewith replaced with the lucid and well-commented Pascal code provided by Mike Fischer.

## 6. Constant of Proportionality Bounds

We now bound the worst-case number of key comparisons and record exchanges performed by *EXTRACT*. Exactly  $n-1$  comparisons are employed to extract the buffer and to count the number of  $L_1$  records. *BLOCKIFY* requires no more than  $n-s-1$  comparisons, all within its while loop. Our selection sort

```

PROCEDURE extract;
VAR
  s: integer;      { block size }
  buffer: integer; { pointer to leftmost buffer element }
  size: integer;   { number of records in the buffer }
  new: integer;    { pointer to next record to be scanned }
  count: integer;  { total number of L1 records }
  i: integer;      { loop index }
BEGIN
  { compute block size }
  s := trunc(sqrt(n));

  { create buffer }
  bufferfill(s, buffer, size);

  IF buffer = 2 THEN
    sort(2, size)
  ELSE BEGIN { buffer size is s }
    { initialize scanner }
    new := buffer - 1;

    { count the number of L1 records }
    count := s + 1;
    FOR i := 2 TO new DO
      IF key(i-1) < key(i) THEN count := count + 1;

    { process remaining records into L1 blocks }
    buffer := blockify(new, count, s);

    { sort buffer }
    sort(buffer, s);

    { put buffer in its proper place }
    blockswap(buffer, count-s+1, s);

    { sort the L1 blocks to right of buffer pointer }
    blocksort(buffer, s, (count+1-buffer) div s - 1);
  END;
END;

{ Fills buffer with s records, if possible. }
PROCEDURE bufferfill(
  s : integer; { size of block }
  VAR buffer : integer; { returns pointer to left end of buffer }
  VAR size : integer { returns size of buffer }
);
BEGIN
  buffer := n+1;
  size := 0;
  REPEAT
    buffer := buffer - 1;
    IF key(buffer-1) = key(buffer) THEN
      { move L2 record to proper place }
      swap(buffer, buffer+size)
    ELSE
      { include L1 record in buffer }
      size := size + 1;
  UNTIL (buffer = 2) or (size = s);
END;

```

implementation of *SORT* can direct at most  $\frac{1}{2}s(s-1)$  comparisons [Kn]. Similarly, *BLOCKSORT* needs no more than  $\frac{1}{2}(t_1-1)(t_1-2) \leq \frac{1}{2}(s+1)(s)$  comparisons. Therefore, the total number of key comparisons is at most  $(n-1) + (n-s-1) + \frac{1}{2}s(s-1) + \frac{1}{2}(s+1)s = 2n + s^2 - s - 2 \leq 3n - s - 2 < 3n$ .

```

[ Processes remaining records. ]
FUNCTION blockify(
  new : integer;      [ pointer to rightmost unprocessed record ]
  count : integer;    [ number of L1-type records in list ]
  s : integer         [ block size ]
) : integer;         [ returns pointer to buffer ]
VAR
  ptr1 : integer;     [ place for next element in current L1 block ]
  ptr2 : integer;     [ place for next element in current L2 block ]
  L2_block : integer; [ pointer to position of next L2 block ]
  el : integer;       [ size of first (partial) block ]

[ Returns a pointer to left end of block containing element i. ]
FUNCTION blockhead(i: integer) : integer;
BEGIN
  blockhead := i - (s + i - el - 1) mod s;
END;

BEGIN (* body of blockify *)
  [ set size of first block ]
  el := count mod s;          [ size of first (partial) block ]

  L2_block := blockhead(new + s); [ where next L2 block goes ]
  ptr1 := L2_block - 1;         [ points into current L1 block ]
  ptr2 := new + s;             [ points into current L2 block ]

  [ fill each L2 block in turn ]
  WHILE L2_block >= count+1 DO BEGIN
    [ fill current L2 block ]
    REPEAT
      [ move past any L1 records ]
      WHILE key(new-1) < key(new) DO BEGIN
        swap(new, ptr1);
        new := new - 1;
        ptr1 := ptr1 - 1;
      END;

      [ found L2 element, so put in current L2 block ]
      swap(new, ptr2);
      new := new - 1;
      ptr2 := ptr2 - 1;
    UNTIL ptr2 + 1 = blockhead(ptr2 + 1);

    [ put current L2 block into proper place at position L2_block ]
    blockswap(ptr2 + 1, L2_block, s);

    [ new L1 block is block containing new ]
    ptr1 := new;

    [ new L2 block is next block to its right ]
    ptr2 := blockhead(new) + 2*s - 1;

    [ swap L1-type elements out of new L2 block ]
    blockswap(new + 1, new + s + 1, ptr2 - new - s);

    [ point to eventual destination of new L2 block ]
    L2_block := L2_block - s;
  END; (* while *)
  blockify := ptr2 - s + 1; [ return pointer to buffer ]
END;

```

As for record exchanges, *SWAP* is invoked at most  $n-s-1$  times. Each *BLOCKSWAP* within *BLOCKIFY* is called at most  $t_2$  times, always with block size less than or equal to  $s$ . The main algorithm's final *SORT* requires

$s-1$  exchanges, followed by *BLOCKSWAP* needing  $s$ . Finally, *BLOCKSORT* uses  $s(t_1-2)$  exchanges. Therefore, the total number of record exchanges is at most  $(n-s-1)+2st_2+(s-1)+s+s(t_1-2)=n+s(t_1+t_2)+st_2-s-2\leq 3n-s-2<3n$ .

Since key comparisons and record exchanges are likely to be by far the most time consuming operations for *EXTRACT*, and since they are both storage-to-storage type instructions for most architectures, we conclude that  $6n$  is a reasonable estimate of the worst-case constant of proportionality for this algorithm's  $O(n)$  time complexity. As for  $O(1)$  space, a review of our code reveals that we have explicitly used but 10 additional storage cells for pointers, counters and the like.

Incidentally, we have conducted a series of experiments to compare *EXTRACT*'s average-case behavior to that of a naive but efficient algorithm free to exploit the temporary use of  $O(n)$  extra memory, and found that the expected penalty for performing duplicate-key extraction in place is less than a quadrupling of program execution times. At the editor's suggestion, however, we have omitted the presentation of our experimental findings from this paper. The interested reader is referred to [HL1].

## 7. Conclusions

We have devised an algorithm that performs stable duplicate-key extraction in linear time and constant space, and is therefore optimal to within a constant factor. We have also bounded its worst-case number of key comparisons and record exchanges to indicate that it is practical.

This algorithm could be especially useful when viewed as an efficient means for increasing the effective size of internal memory when performing duplicate-key extraction on a much larger external file. This tends to decrease the number of input/output operations needed, thereby dramatically reducing the overall execution time. Similarly, this algorithm could be employed to great advantage when managing critical resources such as cache memory or other relatively small, high-speed memory components.

*Acknowledgments.* We wish to express our appreciation to the two anonymous referees, whose thoughtful critiques of our original submission helped improve the readability of this final version. We especially want to thank the editor, Michael J. Fischer, for his useful suggestions on clarifying the presentation in Sect. 4, and for graciously taking the time and interest to implement our algorithm in Pascal.

## References

- [HL1] Huang, B.-C., Langston, M.A.: Stable Duplicate-Key Extraction with Optimal Time and Space Bounds. Proc. 24th Allerton Conf. Commun. Control Comput., pp. 288-295, 1986
- [HL2] Huang, B.-C., Langston, M.A.: Practical In-Place Merging. Commun. ACM 31, 348-352 (1988)
- [HL3] Huang, B.-C., Langston, M.A.: Stable Set and Multiset Operations in Optimal Time and Space. Proc. 7th ACM Symp. Principles Database Syst. pp. 288-293, 1988

- [HL4] Huang, B.-C., Langston, M.A.: Fast Stable Merging and Sorting in Constant Extra Space. Computer Science Department Technical Report CS-87-170, Washington State University, 1987
- [Ho] Horvath, E.C.: Stable Sorting in Asymptotically Optimal Time and Extra Space. *J. ACM* **25**, 177–199 (1978)
- [Kn] Knuth, D.E.: *The Art of Computer Programming*, Vol. 3: Sorting and Searching. Reading, MA: Addison-Wesley 1973
- [Kr] Kronrod, M.A.: An Optimal Ordering Algorithm Without a Field of Operation. *Dok. Akad. Nauk SSSR* **186**, 1256–1258 (1969)
- [MU] Mannila, H., Ukkonen, E.: A Simple Linear-Time Algorithm for In Situ Merging. *Inf. Proc. Lett.* **18**, 203–208 (1984)
- [SS1] Salowe, J.S., Steiger, W.L.: Stable Unmerging in Linear Time and Constant Space. Computer Science Department Technical Report, Rutgers University, 1985
- [SS2] Salowe, J.S., Steiger, W.L.: Simplified Stable Merging Tasks. *J. Algorithms* **8**, 557–571 (1987)
- [Tr] Trabb Pardo, L.: Stable Sorting and Merging with Optimal Space and Time Bounds. *SIAM J. Comput.* **6**, 351–372 (1977)

Received April 2, 1986 / August 22, 1988

## ONLINE VARIABLE-SIZED BIN PACKING\*

Nancy G. KINNERSLEY and Michael A. LANGSTON

*Department of Computer Science, Washington State University, Pullman, WA 99164-1210, USA*

Received 25 February 1987

Revised 27 October 1987

The classical bin packing problem is one of the best-known and most widely studied problems of combinatorial optimization. Efficient offline approximation algorithms have recently been designed and analyzed for the more general and realistic model in which bins of differing capacities are allowed (Friesen and Langston (1986)). In this paper, we consider fast *online* algorithms for this challenging model. Selecting either the smallest or the largest available bin size to begin a new bin as pieces arrive turns out to yield a tight worst-case ratio of 2. We devise a slightly more complicated scheme that uses the largest available bin size for small pieces, and selects bin sizes for large pieces based on a user-specified fill factor  $f \geq 1/2$ , and prove that this strategy guarantees a worst-case bound not exceeding  $1.5 + f/2$ .

### 1. Introduction

In the classical bin packing problem, the objective is to pack a list of  $n$  pieces  $P = \langle p_1, p_2, \dots, p_n \rangle$ , each with a size in the range  $(0, 1]$ , into the minimum number of unit-capacity bins. The general significance of this NP-complete problem is reflected in the great attention it has received in the literature (see [2] for an updated survey).

Recently, important generalizations of the bin packing problem have been investigated [3, 4] in which bin capacities may vary. In particular, the model of [4] permits a fixed collection of bin sizes, where the objective is to minimize the total space of the packing. This model is considerably more realistic than that of the classical problem. (We observe, for example, that the classical problem corresponds to a lumber yard that sells  $2 \times 4$ s in 8-foot lengths only!) In [4], some practical, offline algorithms for variable-sized bin packing were designed and analyzed. The most complicated of those, termed FFDLS, was proved always to produce a packing whose total space is asymptotically bounded by  $\frac{4}{3}$  times the optimum. Also, from a more purely theoretical standpoint, an offline fully polynomial-time approximation scheme has very recently been devised in [11] using a linear programming formulation of the problem.

In this paper, we explore the worst-case behavior of fast, *online* variable-sized bin

\*This research is supported in part by the National Science Foundation under grants ECS-8403859 and MIP-8603879, and by the Office of Naval Research under contract N-00014-88-K-0343.

packing schemes. An online algorithm cannot preview and rearrange the elements of  $P$  before it starts to construct a packing, but must instead accept and immediately pack each piece as it arrives. A number of online strategies have been proposed and analyzed for the classical problem. See, for example, [9, 10, 12, 13]. What makes our problem even more difficult is that whenever an online algorithm decides to begin a new bin, it must also select the size of that bin, and cannot go back later to repack or consolidate bins.

## 2. Notation

Let  $k$  denote the number of distinct bin sizes available, where there is an unlimited supply of bins of each size. We normalize bin and piece sizes so that the largest bin is of size 1 (and thus the size of the largest piece cannot exceed 1). Let  $B = \langle B_1, B_2, \dots, B_l \rangle$  denote the ordered list of  $l$  bins containing  $P$  as packed by an online algorithm, ALG. We use  $B^*$  for the corresponding optimum packing with  $m$  bins. We employ the function  $s$  to specify bin and piece sizes, and use the function  $c$  to specify the total contents of a bin. For example,  $s(p_1)$  denotes the size of the first piece and  $c(B_1)$  denotes the cumulative size of all pieces ALG packs in its first bin. Finally, given an instance  $I$  of variable-sized bin packing, we use  $\text{ALG}(I)$  and  $\text{OPT}(I)$  to denote the values  $\sum_{i=1}^l s(B_i)$  and  $\sum_{i=1}^m s(B_i^*)$ , respectively.

## 3. Some simple algorithms

One option for an online algorithm is simply to begin a new bin whenever the next available piece will not fit into the current bin. If bins of size 1 are always used, this  $O(n)$ -time scheme is called NFL (Next Fit, using Largest possible bins). The following result is from [4] and is reproduced here for the purpose of illustration.

**Theorem 3.1.**  $\text{NFL}(I) < 2 \cdot \text{OPT}(I) + 1$  for any instance  $I$ .

**Proof.** For  $1 \leq i < l$ ,  $c(B_i) + c(B_{i+1}) > 1$ . Therefore

$$\sum_{i=1}^l c(B_i) > \frac{1}{2}(l-1)$$

and

$$\begin{aligned} \text{NFL}(I) &= (l-1) + 1 < 2 \sum_{i=1}^l c(B_i) + 1 \\ &= 2 \sum_{i=1}^m c(B_i^*) + 1 \leq 2 \sum_{i=1}^m s(B_i^*) + 1 = 2 \cdot \text{OPT}(I) + 1. \quad \square \end{aligned}$$

Any packing instance consisting of pieces of size  $\frac{1}{2} + \varepsilon$  and bins of sizes 1 and  $\frac{1}{2} + \varepsilon$ , for some arbitrarily small  $\varepsilon > 0$ , demonstrates that the bound of 2 is asymptotically tight for NFL.

Another alternative is to review all partially packed bins, placing the next available piece in the first bin with room for it, beginning a new bin only when necessary. If bins of size 1 are always used, we denote this approach by FFL (First Fit, using Largest possible bins). By efficiently conducting the review of partially packed bins [6], FFL can be implemented to run in  $O(n \log n)$  time.

**Theorem 3.2.**  $\text{FFL}(I) < 2 \cdot \text{OPT}(I) + 1$  for any instance  $I$ .

**Proof.** Use the same series of arguments presented in the proof of Theorem 3.1.  $\square$

While the worst-case behavior of First Fit is superior to that of Next Fit for the classical problem [2, 5], this is not the case for NFL and FFL when applied to variable-sized bin packing. In fact, *any* online algorithm that uses the largest possible bins will produce the same packing when presented with a troublesome instance such as the one described immediately following the proof of Theorem 3.1.

Given the egregious behavior resulting from the use of large bins, we next consider FFS (First Fit, using Smallest possible bins), of time complexity  $O(n \log n + n \log k)$ .

**Theorem 3.3.**  $\text{FFS}(I) < 2 \cdot \text{OPT}(I) + 1$  for any instance  $I$ .

**Proof.** Since First Fit is used,  $c(B_i) + c(B_i) > s(B_i)$ . Also,  $c(B_i) + c(B_{i+1}) > s(B_i)$  for  $1 \leq i < l$ . Therefore,

$$\begin{aligned}
 \text{FFS}(I) &= \sum_{i=1}^l s(B_i) < s(B_1) + \sum_{i=1}^{l-1} s(B_i) + s(B_l) \\
 &< 2 \sum_{i=1}^l c(B_i) + s(B_l) \\
 &\leq 2 \sum_{i=1}^l c(B_i) + 1 \\
 &= 2 \sum_{i=1}^m c(B_i^*) + 1 \leq 2 \sum_{i=1}^m s(B_i^*) + 1 \\
 &= 2 \cdot \text{OPT}(I) + 1. \quad \square
 \end{aligned}$$

Unfortunately, FFS performs no better in the worst case than does NFL or FFL, since any packing instance consisting of pieces of size  $\frac{1}{2}$  and bins of sizes 1 and  $1 - \varepsilon$ , for some arbitrarily small  $\varepsilon > 0$ , demonstrates that the bound of 2 is asymptotically tight for FFS.

#### 4. Main result

We observe that FFL errs in its packing of “large” pieces (those with size exceeding  $\frac{1}{2}$ ), while FFS errs in its packing of “small” pieces (those with size less than or equal to  $\frac{1}{2}$ ). Therefore, we now focus our attention on a *hybrid* approach [7] that we shall denote by FFf. Let  $f$  denote a user-specified *fill factor* in the range  $[\frac{1}{2}, 1]$ . Suppose FFf must start a new bin using a piece  $p_i$ . If  $p_i$  is a small piece, then FFf starts a new bin of size 1. If  $p_i$  is a large piece, then FFf selects the smallest bin size in the range  $[s(p_i), s(p_i)/f]$  if such a size exists, else it uses bin size 1. For example, if the fill factor is  $\frac{3}{4}$  and a piece,  $p_i$ , needs a new bin, then FFf will select a unit-capacity bin if and only if either  $s(p_i) \leq \frac{1}{2}$  or there is no bin with size less than 1 available that  $p_i$  can fill at least  $\frac{3}{4}$  full.

**Theorem 4.1.**  $\text{FFf}(I) < (1.5 + \frac{1}{2}f) \cdot \text{OPT}(I) + 2$  for any instance  $I$ .

**Proof.** Given an arbitrary instance,  $I$ , we classify bins of the FFf packing as follows: a bin of type  $X$  has size 1 and contains a single piece; a bin of type  $Y$  has size 1 and contains two or more pieces; a bin of type  $Z$  has size less than 1.

We deviate from this classification for at most two “exceptional” bins. Every bin of type  $X$ , except at most one, must contain a large piece. (To see this, observe that if a bin of type  $X$  contains a small piece, then every subsequent bin of type  $X$  must contain a large piece.) If there is a bin of type  $X$  that contains a small piece, then we change its classification from type  $X$  to exceptional. Similarly, every bin of type  $Y$ , except at most one, must be more than  $\frac{2}{3}$  full. (To see this, observe that if a bin of type  $Y$  is at most  $\frac{2}{3}$  full, then every subsequent bin of type  $Y$  must contain at least two pieces, each of size greater than  $\frac{1}{3}$ .) If there is a bin of type  $Y$  that is at most  $\frac{2}{3}$  full, then we change its classification from type  $Y$  to exceptional.

Let  $x$  and  $y$  denote the number of bins of types  $X$  and  $Y$ , respectively. Let  $z$  denote the sum of the sizes of all bins of type  $Z$ . Thus  $\text{FFf}(I) = x + y + z + s$  (any exceptional bins) and we have

$$\text{FFf}(I) - 2 \leq x + y + z. \quad (1)$$

We now obtain two distinct lower bounds for  $\text{OPT}(I)$ . For the first, we consider the bin sizes available. Since every bin of type  $X$  or  $Z$  contains a piece whose size exceeds  $\frac{1}{2}$ , no two such pieces can share one bin in an optimal packing of  $I$ . From the way FFf selects a new bin for a large piece when one is required, a piece,  $p_i$ , from a bin of type  $X$  requires a bin of size at least  $\min\{1, s(p_i)/f\} \geq 1/(2f)$  in an optimal packing. Also, any large piece used by FFf to begin a bin of type  $Z$  is packed as tightly as possible. Therefore, we have

$$\text{OPT}(I) \geq \frac{1}{2}x/f + z. \quad (2)$$

For the second lower bound, we consider the total size of all pieces. The contents of type  $X$  bins sum to more than  $\frac{1}{2}x$ . The contents of type  $Y$  bins sum to more than

$\frac{2}{3}y$ . The contents of type  $Z$  bins sum to at least  $fz$ . Thus, since  $x+y>0$ , we have

$$\text{OPT}(I) > \frac{1}{2}x + \frac{2}{3}y + fz. \quad (3)$$

Let  $R$  denote  $(\text{FFf}(I) - 2)/\text{OPT}(I)$ . From (1) and (2) we know

$$R \leq (x + y + z)/(\frac{1}{2}x/f + z)$$

from which we derive

$$y \geq \frac{1}{2}Rx/f + Rz - x - z. \quad (4)$$

From (1) and (3) we know

$$R < (x + y + z)/(\frac{1}{2}x + \frac{2}{3}y + fz)$$

in which we substitute the lower bound for  $y$  from (4), since it is known [1, 8] that, even for unit-capacity bins, any online algorithm's worst-case ratio exceeds  $1.536 > \frac{3}{2}$ . Thus we derive

$$R < (3x + fx + fz(10 - 6f))/(2x + 4fz)$$

which is bounded above by  $\frac{1}{2}(3x + fx)/x = 1.5 + \frac{1}{2}f$  as long as  $R$  is bounded below by  $\frac{7}{4}$ . Therefore,

$$\text{FFf}(I) = R \cdot \text{OPT}(I) + 2 < (1.5 + \frac{1}{2}f) \cdot \text{OPT}(I) + 2. \quad \square$$

## 5. Discussion

Surprisingly, we conclude from Theorem 4.1 that the simplest variant of FFf may be the best (in the worst-case sense). By setting  $f=0.5$ , a small piece needing a new bin always gets the largest bin available while a large piece needing a new bin always gets the smallest bin that can contain it. Let FFH denote this limiting-case hybrid.

**Corollary 5.1.**  $\text{FFH}(I) < 1.75 \cdot \text{OPT}(I) + 2$  for any instance  $I$ .

For the classical problem, examples exist [5] to demonstrate that  $\text{FF}(I)$  can approach arbitrarily close to  $1.7 \cdot \text{OPT}(I)$  from below. Therefore, of course, the same holds for FFH under the packing model addressed herein.

The determination of just how tight the bounds given in Theorem 4.1 are is as yet an open issue. (Slightly more complicated arguments easily reduce the additive constant from 2 to 1.) However, examples such as the one that follows demonstrate that Corollary 5.1 does not extend to arbitrary values of  $f$ , and that the guarantee of Theorem 4.1 is indeed dependent on  $f$ . Let  $n$  be evenly divisible by 4. Suppose that  $P$  contains  $\frac{1}{2}n$  pieces of size  $\frac{1}{3} + \varepsilon$  followed by  $\frac{1}{2}n$  pieces of size  $\frac{1}{2} + \varepsilon$ , and that bin sizes are  $\frac{2}{3} + 2\varepsilon$  and 1 for some arbitrarily small  $\varepsilon > 0$ . Thus, for  $f=0.6$ ,  $\text{FFf}(I)$  can exceed any value strictly less than  $1.8 \cdot \text{OPT}(I)$ .

Finally, let the parameter  $q$  denote an integer greater than 1 and suppose  $s(p_i) \leq 1/q$  for  $1 \leq i \leq n$ . (In this case, FFF reduces to FFL.) As in [5, Theorem 2.3] we note that FFL insures the inequality  $c(B_i) > q/(q+1)$  for all but at most two values of  $i$  in the range  $[1, l]$ . Therefore, in this parameterized environment, there is no worst-case penalty for variable-sized bin packing. That is, we have

$$\text{FFL}(I) < ((q+1)/q) \cdot \text{OPT}(I) + 2,$$

the same as for the classical problem. However, the freedom to choose bin (and piece) sizes of  $1/(q+1) + \varepsilon$ , for arbitrarily small  $\varepsilon > 0$ , greatly simplifies the job of establishing the asymptotic tightness of this parameterized bound.

## References

- [1] D.J. Brown, A lower bound for on-line one-dimensional bin packing algorithms, Tech. Rept. R-864, Coordinated Science Laboratory, University of Illinois, Urbana, IL (1979).
- [2] E.G. Coffman, Jr., M.R. Garey and D.S. Johnson, Approximation algorithms for bin packing: An updated survey, in: G. Ausiello, M. Lucertini and P. Serafini, eds., *Algorithm Design for Computer Systems Design* (Springer, New York, 1984), 49–106.
- [3] D.K. Friesen and M.A. Langston, A storage size selection problem, *Inform. Process. Lett.* 18 (1984) 295–296.
- [4] D.K. Friesen and M.A. Langston, Variable sized bin packing, *SIAM J. Comput.* 15 (1986) 222–230.
- [5] D.S. Johnson, A. Demers, J.D. Ullman, M.R. Garey and R.L. Graham, Worst-case performance bounds for simple one-dimensional packing algorithms, *SIAM J. Comput.* 3 (1974) 299–325.
- [6] D.S. Johnson, Fast algorithms for bin packing, *J. Comput. Syst. Sci.* 8 (1974) 272–314.
- [7] M.A. Langston, A study of composite heuristic algorithms, *J. Oper. Res. Soc.* 38 (1987) 539–544.
- [8] F.M. Liang, A lower bound for on-line bin packing, *Inform. Process. Lett.* 10 (1980) 76–79.
- [9] C.C. Lee and D.T. Lee, A simple on-line bin packing algorithm, *J. ACM* 32 (1985) 562–572.
- [10] C. Martel, A linear on-line bin packing algorithm, Tech. Rept. CSRL-83-12, Department of Electrical and Computer Engineering, University of California, Davis, CA (1983).
- [11] F.D. Murgolo, An efficient approximation scheme for variable-sized bin packing, *SIAM J. Comput.* 16 (1987) 149–161.
- [12] P. Ramanan and D.J. Brown, On-line bin packing in linear time, Tech. Rept. R-1003, Coordinated Science Laboratory, University of Illinois, Urbana, IL (1983).
- [13] A.C. Yao, New algorithms for bin packing, *J. ACM* 27 (1980) 207–227.

## RESOURCE ALLOCATION UNDER LIMITED SHARING

Michael A. LANGSTON\*

*Department of Computer Science, University of Tennessee, Knoxville, TN 37996-1301, USA; and  
Department of Computer Science, Washington State University, Pullman, WA 99164-1210, USA*

Michael P. MORFORD

*NCR Corporation, 9900 Old Grove Road, San Diego, CA 92131, USA*

Received 15 March 1988

Revised 30 January 1989

We study the problem of resource sharing within a system of users, each with the same resource capacity, but with varying resource demands. For model simplicity, we assume system saturation, so that the total demand matches the total capacity, and permit only a limited form of sharing in which a user is free to share its unused capacity with exactly one other user. We seek to maximize the total amount of unshared capacity over all feasible solutions, reflecting an environment in which sharing incurs a cost proportional to the overall quantity shared. For the general problem, which is NP-hard, we derive a tight worst-case performance bound for a greedy algorithm  $G$  as well as for a number of other sharing rules. We also prove several results concerning  $G$ 's behavior in more restricted settings.

**Keywords.** Resource allocation, computational complexity, approximation algorithms, combinatorial optimization.

### 1. Introduction

Consider a collection of  $n$  users, each with an identical resource capacity  $C$ , and each with a specific resource demand  $d_i$ ,  $1 \leq i \leq n$ . The simplest case, and the one on which we focus our attention, assumes that the system is saturated, so that  $\sum_{i=1}^n d_i = nC$ . A user whose demand is less than  $C$  is permitted to *share* its excess capacity with one whose demand exceeds  $C$ . We seek to redistribute such excess within the system so as to maximize the total capacity unshared, modeling an environment in which the cost of sharing is directly proportional to the overall quantity shared.

If the number of users allowed to share one user's excess is unlimited, then this problem can be easily solved in polynomial time and is left to the reader as an exercise. If a bound is placed on the number of users who may simultaneously share one user's excess, then the problem is NP-hard. The extreme case, and the one we explore here, permits a very *limited* form of sharing, in which only one user is permitted

\* This author's research has been supported in part by the National Science Foundation under grants ECS-8403859, MIP-8603879 and MIP-8919312, and by the Office of Naval Research under contract N00014-88-K-0343.

to share another's excess. (This limit applies to the lender's excess, not the borrower's demand. A user with a great demand may have to borrow from several other users. Moreover, a borrower whose demand is strictly less than  $C$  plus the capacity he must borrow subsequently becomes a lender himself.) As we shall show, even this primitive formulation is combinatorially rich and exceedingly difficult to optimize.

This model can be interpreted in several ways. It was first brought to our attention as a problem of efficiently generating random choices from a finite set [5]. Here, the well-known "square-the-histogram" method corresponds to the limited sharing problem just described, with the objective to maximize the expected number of direct memory accesses. (A uniformly generated pseudo-random number that falls in a shared region corresponds to a unique but secondary choice, thus requiring an indirect memory fetch. See [9, 10] for more details.) One can also visualize the model as representative of a distributed computing environment in which the local memory of a processing element can share unused storage with other elements. Limited I/O porting, communications overhead, memory addressing restrictions (e.g., bounds registers) and a host of other hardware and software limitations can severely inhibit sharing. Our problem can even be viewed as one of resource balancing, a one-dimensional analog of the problem of evenly distributing goods among a collection of warehouses, where transportation costs or other factors dictate that excess space must be occupied by goods coming from only one other warehouse. Perhaps the most superficially similar, previously-studied problem is that of variable sized bin packing [2], where a bin is akin to a demand exceeding  $C$ , and a piece to be packed is much like a user's excess capacity.

The remainder of this paper is organized as follows. In the next section, we introduce some necessary notation, state the decision version of this problem, and demonstrate that it is NP-complete. We also define a natural greedy rule  $G$  whose analysis makes up the main thrust of this study. Section 3 contains our proof that, for the general case,  $G$ 's solution always exceeds half the optimum. We devise a powerful, easy-to-apply *chain lemma* to aid in the analysis and show, through a family of problem instances, that this bound is asymptotically tight. Furthermore, we discuss a number of sharing alternatives and demonstrate that (in the worst-case sense) these appealing but more complicated rules do no better than  $G$ . In Section 4, we consider  $G$ 's behavior in more restricted settings. For some (most notably, when all demands exceeding  $C$  are equal and all demands less than  $C$  are equal), we prove that  $G$  is optimal. We address research directions and related issues in the closing section, showing that this problem is so difficult that when it is generalized slightly by allowing users to possess unequal resource capacities, the problem of determining whether *any* feasible solution exists is NP-complete.

## 2. Notation and preliminaries

Without loss of generality, we assume a scaling so that the common resource

capacity  $C$  is a positive integer, and so that the elements of the initial demand list  $D^{(0)} = (d_1^{(0)}, d_2^{(0)}, \dots, d_n^{(0)})$  are nonnegative integers, where the quantity  $d_i^{(0)}$ ,  $1 \leq i \leq n$ , denotes the initial demand of user  $i$ . For simplicity, we assume a user indexing whereby the initial demand list is sorted so that  $d_1^{(0)} \geq d_2^{(0)} \geq \dots \geq d_n^{(0)}$ . We dynamically alter the demand list as we implement limited sharing, requiring that, at any time  $t$ ,  $D^{(t)}$  denotes the current demand list, with  $d_i^{(t)}$  representing the current demand of user  $i$ .

Let  $A^{(t)}$  denote the list made up of the elements of  $D^{(t)}$  that are greater than or equal to  $C$ . Let  $|A^{(t)}|$  denote the number of elements in  $A^{(t)}$ . Similarly, let  $B^{(t)}$  denote  $D^{(t)} - A^{(t)}$  (that is,  $B^{(t)}$  contains every element of  $D^{(t)}$  that is less than  $C$ ), where  $|B^{(t)}|$  denotes the number of elements in  $B^{(t)}$ . Thus  $A^{(0)}$  ( $B^{(0)}$ ) is a prefix (suffix) of  $D^{(0)}$ .

To define limited sharing formally, we now describe how  $D^{(t)}$  (and, hence,  $A^{(t)}$  and  $B^{(t)}$ ) may be altered in an effort to satisfy all demands. As long as  $|B^{(t)}| > 0$ , we denote a *sharing action* by the ordered pair of indices  $(a, b)$ , where  $d_a^{(t)} > C$  and  $d_b^{(t)} < C$ . For notational convenience, we associate one sharing action with the passage of one time unit. The effect of a sharing action at time  $t + 1$  is to transform  $D^{(t)}$  into  $D^{(t+1)}$  by replacing  $d_a^{(t)}$  with  $d_a^{(t+1)} = d_a^{(t)} + d_b^{(t)} - C$  and replacing  $d_b^{(t)}$  with  $d_b^{(t+1)} = C$ . Thus a sharing action directs user  $b$  to allocate all of its excess capacity to user  $a$ .

Note that a sharing action is possible whenever  $|B^{(t)}| > 0$ . Moreover, such an action  $(a, b)$  denies user  $b$  the opportunity to participate in a subsequent action. Therefore, it is elementary that a series of at most  $n - 1$  sharing actions produces a feasible solution, forcing every demand to converge on  $C$ . We term such a series a *sharing sequence*.

For a sharing sequence  $S$  with  $k < n$  sharing actions (which by definition produces  $D^{(k)}$  such that  $d_1^{(k)} = d_2^{(k)} = \dots = d_n^{(k)} = C$ ), we define the *unshared capacity of user  $i$*  to be the minimum value in the set  $\{d_i^{(t)} : 0 \leq t \leq k\}$ . Notice that this value is exactly  $C$  if and only if the demand of user  $i$  was represented in  $A^{(t)}$  at every time  $t$ ,  $0 \leq t \leq k$ . Given  $D^{(0)}$  and an applicable sharing sequence  $S$ , we define the *total unshared capacity* in the obvious way, as the sum of every user's unshared capacity.

In the sequel, we shall restrict our attention to solution strategies that operate by producing a sharing sequence. That is, such a strategy must direct that (1) user  $b$  can lend only when its demand is less than  $C$  and (2) user  $a$  can borrow only when its demand is greater than  $C$ . Condition (1) is merely a convenience, since a user can lend only once. If we first compute the amount to be lent and to whom it goes, the time of the relevant sharing action is immaterial. Condition (2), on the other hand, is significant since a user can borrow repeatedly, and our restriction is justified as follows.

**Theorem 2.1.** *An optimal solution cannot direct that excess capacity be allocated to a user whose current demand is less than or equal to  $C$ .*

**Proof.** Suppose otherwise for an optimal solution that, at some time  $t$ , directs user

$j$  with  $d_j^{(t-1)} < C$  to allocate its excess to user  $i$  with  $d_i^{(t-1)} \leq C$ . Without loss of generality, suppose the optimization rule next directs user  $i$  to allocate its excess to some user  $h$ . Then  $d_h^{(t+1)} = d_h^{(t)} + d_i^{(t)} - C = d_h^{(t)} + d_i^{(t-1)} + d_j^{(t-1)} - 2C$ , since the unshared capacity of user  $i$  is  $d_i^{(t)} = d_i^{(t-1)} + d_j^{(t-1)} - C$ , and that of user  $j$  is simply  $d_j^{(t-1)}$ . In this situation, we can modify the solution by directing user  $j$  to allocate its excess directly to user  $h$  at time  $t$ , then next directing user  $i$  to allocate its excess to user  $h$  as well. This modification preserves  $d_h^{(t+1)} = d_h^{(t)} + d_i^{(t-1)} + d_j^{(t-1)} - 2C$ . The unshared capacity of user  $j$  is still  $d_j^{(t-1)}$ . But now the unshared capacity of user  $i$  is  $d_i^{(t-1)} > d_i^{(t-1)} + d_j^{(t-1)} - C$ , contradicting the presumption that the original solution maximized the unshared capacity.  $\square$

We now address the complexity of limited sharing with the following decision version of the problem.

#### LIMITED SHARING PROBLEM (LSP)

*Input.* A positive integer  $Q$  and a list  $L$  of  $n$  nonnegative integers.

*Question.* Is there a sharing sequence, using  $D^{(0)} = L$  sorted in nonincreasing order and using  $C = \sum_{i=1}^n d_i^{(0)} / n$ , for which the total unshared capacity is at least  $Q$ ?

This problem has already been shown to be NP-complete [9]. We shall present our simple proof of its complexity here for the purpose of illustration and because it will be referenced and built upon in proving subsequent results.

**Theorem 2.2.** LSP is NP-complete.

**Proof.** LSP is clearly in NP, since a candidate solution can be easily checked in polynomial time. To show that LSP is NP-hard, transform any instance of PARTITION [4] into an instance of LSP as follows. Let the list of integers input to PARTITION be denoted by  $p = (p_1, p_2, \dots, p_m)$ , where  $p_1 \geq p_2 \geq \dots \geq p_m$ . Let  $s$  denote  $\sum_{i=1}^m p_i$ . Set  $n = m + 2$ , set  $Q = np_1 - s$ , and set  $L = (l_1, l_2, \dots, l_n)$ , where  $l_1 = l_2 = p_1 + s/2$  and  $l_{i+2} = p_1 - p_{m-i+1}$  for  $1 \leq i \leq m$ . The answer to this instance of LSP is “yes” if and only if the answer to the given instance of PARTITION is “yes”.  $\square$

Thus we conclude that finding a sharing sequence that maximizes the total unshared capacity is NP-hard. Hence we turn our attention to the design and analysis of fast approximation algorithms. For a heuristic algorithm ALG and an optimization algorithm OPT, we use  $\text{ALG}(I)$  and  $\text{OPT}(I)$  to denote their respective total unshared capacities for LSP instance  $I$ . We seek to establish a *worst-case ratio*  $R_{\text{ALG}}$ , defined as follows:

$$R_{\text{ALG}} = \sup \left\{ \frac{\text{OPT}(I)}{\text{ALG}(I)} : I \text{ is an instance of LSP} \right\}.$$

If we can prove that  $R_{\text{ALG}}$  is bounded above by some constant, then we say that ALG is a *relative approximation algorithm*.

For notational convenience, we shall also employ the term  $\text{ALG}_A(I)$  to denote the contribution to  $\text{ALG}(I)$  made by the users whose demands were represented in  $A^{(0)}$  for instance  $I$ . We define  $\text{ALG}_B(I)$  analogously for  $B^{(0)}$ , so that  $\text{ALG}(I) = \text{ALG}_A(I) + \text{ALG}_B(I)$ .

At every time  $t$ , our greedy rule  $G$  (dubbed the ‘‘Robin Hood’’ rule in [10]), simply selects  $a$  so as to maximize  $d_a^{(t)}$  and  $b$  so as to minimize  $d_b^{(t)}$ . That is, it iteratively directs that user  $b$ , currently with the largest excess capacity, lend it all to user  $a$ , currently with the greatest demand. Ties are broken in favor of the user with the lower index. The time complexity of  $G$  is  $O(n \log n)$ , since  $O(n \log n)$  time is sufficient for the initial sorting of the  $n$  demands, and since the values for  $a$  and  $b$  during each of  $G$ 's at most  $n - 1$  sharing actions can be determined in  $O(\log n)$  time with the use of a max-heap for  $A$  and a min-heap for  $B$ .

### 3. The general case

The main goal of this section is to establish the exact value of  $R_G$ . We first demonstrate that  $R_G$  exceeds any real number strictly less than 2.

**Example 3.1.** A troublesome instance  $I_k$  for  $G$ .

Let  $k$  denote any integer exceeding 1 and let  $n = 3k - 1$ .

Let  $D^{(0)}$  be defined as follows:

$$\begin{aligned} d_1^{(0)} &= k^2 + k + 1, \\ d_i^{(0)} &= 2k + 2, & \text{for } 1 < i \leq k, \\ d_i^{(0)} &= 1, & \text{for } k < i \leq 2k, \\ d_i^{(0)} &= 0, & \text{for } 2k < i \leq 3k - 1. \end{aligned}$$

Thus  $C = k + 1$ , and

$$R_G \geq \frac{\text{OPT}(I_k)}{G(I_k)} = \frac{k(k+2)}{k(k+5)/2} = 2 - \frac{6}{k+5},$$

which approaches arbitrarily close to 2 from below as  $k$  grows without bound.

In Example 3.1, observe that  $G$ 's first  $k - 1$  sharing actions distribute most of  $d_1^{(0)}$  across the last  $k - 1$  users, and so  $d_1^{(k-1)} = k^2 + k + 1 - (k - 1)(k + 1) = k + 2$ .  $G$ 's next  $k - 1$  sharing actions produce  $d_i^{(2k-2)} = k + 2$ , for  $1 \leq i \leq k$ , and  $B^{(2k-2)} = (d_{2k}^{(2k-2)} = 1)$ . Therefore  $G_A(I_k) = \sum_{i=2}^{k+1} i = k(k+3)/2$ . Since  $G_B(I_k) = k$ , we conclude that  $G(I_k) = k(k+3)/2 + k = k(k+5)/2$ . A better solution can be devised by first directing that the unsatisfied demand of user  $i$  be allocated the excess capacity of user  $j = i + 2k - 1$ , for  $1 < i \leq k$ , from which it follows that  $\text{OPT}(I_k) = k(k+1) + k = k(k+2)$ .

Thus we know that 2 is a lower bound on  $R_G$ . We now proceed to prove that 2 is an upper bound as well.

**Lemma 3.2.** *If  $\text{ALG}_A(I) > |A^{(0)}|C/2$ , then  $\text{OPT}(I) < 2\text{ALG}(I)$ .*

**Proof.** Clearly,  $\text{OPT}_A(I) \leq |A^{(0)}|C$ . Any algorithm,  $\text{ALG}$ , that produces a sharing sequence in an attempt to solve LSP insures that  $\text{ALG}_B(I)$  is precisely the total demand in  $B^{(0)}$ , and hence  $\text{ALG}_B(I) = \text{OPT}_B(I)$ . Thus it follows that, if  $\text{ALG}_A(I) > |A^{(0)}|C/2$ , then  $\text{OPT}(I) = \text{OPT}_A(I) + \text{OPT}_B(I) \leq |A^{(0)}|C + \text{ALG}_B(I) < 2\text{ALG}_A(I) + \text{ALG}_B(I) \leq 2\text{ALG}(I)$ .  $\square$

In order to state and prove the next lemma, it will be helpful first to define a *chain* of users, each of whose demands were represented in  $A^{(0)}$ , with respect to a sharing sequence  $S$ . We denote such a chain of length  $k$  by the ordered list of indices  $U = (u_1, u_2, \dots, u_k)$ , where  $1 \leq u_h \leq |A^{(0)}|$  for  $1 \leq h \leq k$ . At some time  $t_1 \geq 0$ , the first member of such a chain,  $u_1$ , must occur in a sharing action of  $S$  of the form  $(a_1 = u_1, b_1)$ , where  $d_{b_1}^{(0)}$  is in  $B^{(0)}$  and  $d_{a_1}^{(t_1)} < C$ . (Thus  $d_{b_1}^{(0)} = d_{b_1}^{(t_1-1)} < C$ , and  $d_{b_1}^{(t_1)} = C$ .) If, at some time  $t_2 > t_1$ ,  $u_1$  occurs in a sharing action of the form  $(a_2, b_2 = u_1)$  where  $d_{a_2}^{(t_2)} < C$ , then  $u_2 = a_2$  is the second user in the chain, and so on. The chain ends with  $u_k$  when, at time  $t_{k+1}$ ,  $u_k$  occurs in a sharing action of the form  $(a_{k+1}, b_{k+1} = u_k)$ , where  $d_{a_{k+1}}^{(t_{k+1})} \geq C$ .

**Lemma 3.3** (the chain lemma). *If  $H$  is an LSP heuristic that produces a sharing sequence in which, at every given time  $t$ ,  $a$  is selected so as to maximize  $d_a^{(t)}$ , then the contribution to  $H_A(I)$  made by the users in a user chain of length  $k$  is at least  $kC/2$ .*

**Proof.** Let  $H$  satisfy the statement of the lemma. Let  $I$  denote an instance of LSP with chain  $U = (u_1, u_2, \dots, u_k)$  for some  $k > 0$ . Since  $H$  produces a sharing sequence, the contribution to  $H_A(I)$  made by the users in  $U$  is  $\sum_{h=1}^k d_{u_h}^{(t_h)}$ . By  $H$ 's choice of  $a$  for each sharing action,  $d_{u_h}^{(t_h-1)} \geq d_{u_{h+1}}^{(t_h-1)}$  for  $1 \leq h < k$ . Therefore,  $d_{u_1}^{(t_1)} \geq d_{u_2}^{(t_2)} - d_{u_1}^{(t_1)}$ . Also,  $d_{u_h}^{(t_h)} - d_{u_{h-1}}^{(t_h-1)} \geq d_{u_{h+1}}^{(t_h)} - d_{u_h}^{(t_h)}$  for  $1 < h < k$ , and  $d_{u_k}^{(t_k)} - d_{u_{k-1}}^{(t_k-1)} \geq C - d_{u_k}^{(t_k)}$ . From this set of  $k$  inequalities we obtain  $d_{u_1}^{(t_1)} \geq C/(k+1)$ , else we derive  $C = d_{u_1}^{(t_1)} + (d_{u_2}^{(t_2)} - d_{u_1}^{(t_1)}) + \dots + (C - d_{u_k}^{(t_k)}) < (k+1)C/(k+1) = C$ , which is impossible. Similarly, we obtain  $d_{u_h}^{(t_h)} \geq hC/(k+1)$  for  $1 < h \leq k$ . Hence we conclude that  $\sum_{h=1}^k d_{u_h}^{(t_h)} \geq \sum_{h=1}^k hC/(k+1) = (k(k+1)/2)C/(k+1) = kC/2$ .  $\square$

We now establish the general utility of the chain lemma.

**Theorem 3.4.** *If  $H$  is an LSP heuristic that produces a sharing sequence in which, at every time  $t$ ,  $a$  is selected so as to maximize  $d_a^{(t)}$ , then  $R_H \leq 2$ .*

**Proof.** Given such an  $H$  and any instance  $I$  of LSP, no user can be represented in more than one user chain. Moreover, there must always exist at least one user from

$A^{(0)}$  that is not represented in any chain. We apply the chain lemma to every chain in  $I$ , guaranteeing that  $H_A(I) \geq (|A^{(0)}| - 1)C/2 + C > |A^{(0)}|C/2$ , from which Lemma 3.2 tells us that  $\text{OPT}(I)/A(I) < 2$ . Therefore, by the definition of worst-case ratio,  $R_H \leq 2$ .  $\square$

**Corollary 3.5.**  $R_G = 2$ .

**Proof.** Follows immediately from Example 3.1 and Theorem 3.4.  $\square$

Thus  $G$  is a relative approximation algorithm. That LSP permits such an algorithm is interesting, since some NP-hard problems do not (see, for example, the non-Euclidean traveling salesman problem [8] or the weighted depth-first spanning tree problem [1]).

Are there relative approximation algorithms with lower worst-case ratios? A second alternative algorithm, which we denote by  $A_2$ , selects  $a$  as does  $G$  so as to maximize  $d_a^{(t)}$ , but selects  $b$  so as to maximize  $d_b^{(t)}$  as well. Thus  $A_2$  is nearly optimal for Example 3.1. Unfortunately,  $A_2$  asymptotically fares no better than  $G$  in the worst case.

**Example 3.6.** A troublesome instance  $I_k$  for  $A_2$ .

Let  $k$  denote any integer exceeding 1 and let  $n = k^2 + 2k + 2$ .

Let  $D^{(0)}$  be defined as follows:

$$\begin{aligned} d_i^{(0)} &= k^2 + 2k + 1, & \text{for } 1 \leq i \leq k + 1, \\ d_i^{(0)} &= 1, & \text{for } k + 1 < i \leq 2k + 2, \\ d_i^{(0)} &= 0, & \text{for } 2k + 2 < i \leq k^2 + 2k + 2. \end{aligned}$$

Thus  $C = k + 1$ , and

$$R_{A_2} \geq \frac{\text{OPT}(I_k)}{A_2(I_k)} = \frac{(k+1)(k+2)}{(k+1)(k+4)/2} = 2 - \frac{4}{k+4},$$

which approaches arbitrarily close to 2 from below as  $k$  grows without bound.

In Example 3.6, observe that an optimal sharing sequence can be devised by first directing that the unsatisfied demand of user 1 be allocated the excess capacity of the  $k + 1$  users with indices in the range  $[k + 2, 2k + 2]$ .

**Corollary 3.7.**  $R_{A_2} = 2$ .

**Proof.** Follows immediately from Example 3.6 and Theorem 3.4.  $\square$

A third alternative  $A_3$  selects  $a$  so as to minimize  $d_a^{(t)} > C$  and  $b$  so as to minimize  $d_b^{(t)}$  (in which case the chain lemma does not apply). Thus  $A_3$  is optimal for both

Examples 3.1 and 3.6. However,  $A3$  fails to certify as even a relative approximation algorithm.

**Example 3.8.** A troublesome instance  $I_k$  for  $A3$ .

Let  $k$  denote any integer exceeding 1 and let  $n = 2k + 1$ .

Let  $D^{(0)}$  be defined as follows:

$$\begin{aligned} d_1^{(0)} &= k^2, \\ d_i^{(0)} &= k + 1, \quad \text{for } 1 < i \leq k + 1, \\ d_i^{(0)} &= 0, \quad \text{for } k + 1 < i \leq 2k + 1. \end{aligned}$$

Thus  $C = k$ , and

$$R_{A3} \geq \frac{\text{OPT}(I_k)}{A3(I_k)} = \frac{k(k+3)/2}{2k} > \frac{k}{4},$$

which is unbounded above.

A fourth alternative  $A4$  selects  $a$  so as to minimize  $d_a^{(t)}$  and  $b$  so as to maximize  $d_b^{(t)}$ . Like  $A3$ ,  $A4$  is not a relative approximation algorithm.

**Example 3.9.** A troublesome instance  $I_k$  for  $A4$ .

Let  $k$  denote any integer exceeding 1 and let  $n = k + 2$ .

Let  $D^{(0)}$  be defined as follows:

$$\begin{aligned} d_1^{(0)} &= 2k^2 - k, \\ d_i^{(0)} &= k^2 + 1, \quad \text{for } 1 < i \leq k + 1, \\ d_{k+2}^{(0)} &= 0. \end{aligned}$$

Thus  $C = k^2$ , and

$$R_{A4} \geq \frac{\text{OPT}(I_k)}{A4(I_k)} = \frac{k(k+1)(k-1/2)}{k(3k+1)/2} > \frac{k}{2},$$

which is unbounded above.

What about a “best fit” strategy of some sort? For example, such an alternative  $A5$  could initially choose  $a$  and  $b$  as  $G$  would, but then, if  $d_a^{(t)} - C > C - d_b^{(t)}$ , reselect  $a$  so that  $d_a^{(t)}$  is a least demand satisfying  $d_a^{(t)} - C \geq C - d_b^{(t)}$ . But this scheme is asymptotically no better than  $G$ . To see this, simply modify Example 3.1 by setting  $d_i^{(0)} = 2k + 1$  for  $1 < i \leq k$  and  $d_i^{(0)} = 2$  for  $k < i \leq 2k - 1$ , in which case  $A5$  reduces to  $G$ .

**Corollary 3.10.**  $R_{A5} = 2$ .

**Proof.** Follows immediately from the modification to Example 3.1 described above

and the observation that the chain lemma (and hence Theorem 3.4) holds as long as  $H$  maximizes  $d_a^{(t)}$  whenever it must start a new chain or append to an existing chain, at which time  $d_a^{(t)} - C < C - d_b^{(t)}$ .  $\square$

Finally, what of “compound” algorithms (those that implement multiple heuristics and select the best solution produced)? Although this approach may work well in practice, it might not yield improved worst-case behavior or its analysis might be exceedingly difficult (proofs of improved worst-case behavior for compound algorithms are very rare [3, 6]). To illustrate, consider alternative  $A_6$ , a compound algorithm running both  $G$  and  $A_2$ .

**Corollary 3.11.**  $R_{A_6} = 2$ .

**Proof.** Follows immediately from Example 3.6 (although the lower bound thereby obtained for  $G$  is neither as simple nor as fast-growing as the one exhibited in Example 3.1) and either Corollary 3.5 or 3.7.  $\square$

As of this writing, we know of no polynomial-time algorithm  $ALG$  with  $R_{ALG} < 2$ , and suspect that guaranteeing a bound strictly less than 2 may be NP-hard.

#### 4. Special cases of LSP

In this section, we investigate  $G$ 's behavior in more restricted problem settings. We know from Example 3.1 that  $G$  may not guarantee an optimal solution when  $n \geq 5$ .

**Theorem 4.1** [7]. *If  $I$  denotes any instance of LSP with  $n \leq 4$ , then  $G(I) = \text{OPT}(I)$ .*

Suppose all demands greater than  $C$  are alike, as are all demands less than  $C$ .

**Theorem 4.2.** *If  $I$  denotes any instance of LSP with  $d_i^{(0)} = p \geq C$  or  $d_i^{(0)} = q < C$  for every  $i \in [1, n]$ , then  $G(I) = \text{OPT}(I)$ .*

**Proof.** Suppose otherwise, and let  $I$  denote a counterexample with  $|D^{(0)}| = n$ . Without loss of generality, we assume that  $I$  is minimal. That is, no counterexample exists with fewer than  $n$  demands. Since, by assumption,  $D^{(0)}$  is sorted in nonincreasing sequence, there exists a unique  $k$ , where  $1 \leq k < n$ , such that  $d_1^{(0)} = d_2^{(0)} = \dots = d_k^{(0)} = p$  and  $d_{k+1}^{(0)} = d_{k+2}^{(0)} = \dots = d_n^{(0)} = q$ .

Let us now scrutinize an optimal sharing sequence  $S$  to discover how it can provide a greater total unshared capacity than does  $G$ .  $S$  must contain exactly  $n - k$  sharing actions of the form  $(a, b)$ , with  $b > k$ . Since the capacity shared in each of these actions is fixed at  $C - q$ , we can for simplicity assume that these  $n - k$  actions occur first in  $S$ . This implies that  $S$  must distribute the excess capacity in  $B^{(0)}$  (which

is  $(n-k)(C-q)$  in a manner fundamentally different from that of  $G$ , else the two sublists  $A^{(n-k)} - B^{(0)}$  and  $B^{(n-k)}$  constitute a counterexample with  $k < n$  demands, violating the presumed minimality of  $I$ . Moreover, some demand represented in  $A^{(n-k)}$  must exceed  $C + (C-q) = 2C - q$ , since otherwise, to be different from  $G$ ,  $S$  would have to leave some user in  $B^{(n-k)}$  with an unshared capacity less than  $C - (C-q) = q$ , which is impossible for a sharing sequence.

We shall now prove that  $S$  is not optimal, thereby deriving a contradiction. Let  $g$  denote the index of an arbitrary user with  $d_g^{(n-k)} > 2C - q$ . From the set of sharing actions (after time  $n-k$ ) of the form  $(g, b \leq k)$ , select  $t$  so as to maximize  $d_g^{(t-1)} - d_g^{(t)}$ . Define  $h$  as the lender at time  $t$ . That is, the action at time  $t$  is  $(g, h)$ .

We insist that  $(g, h)$  be the last sharing action with  $g$  as the borrower. If this is not the case, we modify  $S$  at no cost as follows. Define  $t_l$  as the time of the last action of the form  $(g, b)$ . We delay the action  $(g, h)$  until time  $t_l$ , and advance by one time unit the subsequence of  $t_l - t$  actions originally scheduled to begin at time  $t+1$ . After this, we reset  $t$  to  $t_l$ . By our choice of  $h$ , this modification produces a sharing sequence, with no loss in the total unshared capacity.

Recalling the definition of a user chain as presented in Section 3, we observe that  $S$  contains a chain  $U = (u_1, u_2, \dots, u_k)$  such that for some  $i$ ,  $1 \leq i \leq k$ ,  $u_i = h$  and thus  $t_{i+1} = t$ . If  $i = k$ , then we know that  $d_g^{(t)} = C$ . If  $i < k$ , then we know that  $u_{i+1} = g$  and  $d_g^{(t)} < C$ . Let  $t' > n - k \geq t_1$  denote the earliest time at which  $d_g^{(t')} \leq 2C - q$ . For future reference, let  $x = C - d_h^{(t-1)} > 0$  and  $y = C - d_g^{(t)} \geq 0$ . Note that  $x > y$ .

We now modify  $S$  as follows. First, we change the sharing action at time  $t_1$  from  $(u_1, b > k)$  to  $(g, b)$ . Next, we change every sharing action of the form  $(g, b \leq k)$  that occurs after time  $t'$  but before time  $t$  to  $(u_1, b)$ . We delete the action  $(g, h)$  at time  $t$ . We insert in its place the action  $(u_1, g)$ , but only if  $d_g^{(t-1)}$  is now less than  $C$ . Finally, we check to see if there is a subsequent action of the form  $(f, g)$  and, if so, delete it, inserting in its place the action  $(f, h)$ . In the new sequence, let  $z = C - d_g^{(t')} \geq 0$ . Note that, by our choice of  $h$  and  $t'$ ,  $x > z$ .

Consider the effect of this modification. To the unshared capacity of user  $g$ , we have added  $y - z$ , which may reflect an increase or a decrease. To the unshared capacity of  $u_1$  (and hence also to the unshared capacity of each user  $u_j$ ,  $1 < j \leq i$ ), we have added  $x - y$ , which is an increase, and which is small enough not to halt  $U$  prematurely. The unshared capacity of all other users, including any that follow  $g$  in  $U$ , is unaltered. Therefore, the total unshared capacity is increased by  $(y - z) + i(x - y) = x - z + (i - 1)(x - y) > 0$ . (Furthermore, the actions at user  $u_1$  may not now even constitute a sharing sequence, in which case a further increase may be obtainable.) We conclude that  $S$  as originally given was not optimal, a contradiction.  $\square$

Suppose only the demands greater than (less than)  $C$  are equal. Let  $LSP_A$  ( $LSP_B$ ) denote all problem instances under this restriction.

**Corollary 4.3.**  $LSP_A$  is NP-complete.

**Proof.** Follows immediately from the construction used in the proof of Theorem 2.2.  $\square$

**Corollary 4.4.** *When restricted to instances of  $LSP_A$ ,  $R_G=2$ .*

**Proof.** Follows immediately from Theorem 3.4 and from observing  $G$ 's behavior when applied to the family of instances defined in Example 3.6.  $\square$

**Theorem 4.5.**  $LSP_B$  is NP-complete.

**Proof.** Modify the proof of Theorem 2.2 as follows. Set  $n=2m+2$ . Set  $Q=2ms+3s$ . Set  $l_1=l_2=5s/2$ . Set  $l_{i+2}=4s-p_i$  for  $1 \leq i \leq m$  and  $l_i=0$  for  $m+2 \leq i \leq 2m+2$ . (Thus  $C=2s$ , and all demands less than  $C$  are equal. By this choice of  $Q$ , there can be no sharing action of the form  $(a, b)$  where  $a=1$  or  $2$  and  $b > m+2$ , so that the first  $m$  sharing actions must absorb all the excess capacity in  $B^{(0)}$ .) The answer to this instance of  $LSP_B$  is "yes" if and only if the answer to the given instance of PARTITION is "yes".  $\square$

Although  $R_G$  for  $LSP_B$  cannot exceed 2 (by Theorem 3.4), its exact value is an open issue as of this writing. We conjecture that it is strictly less than 2. To justify this sentiment, we observe that our worst examples for  $G$  depend on  $G$  at some point driving a demand in  $A$  nearly to zero, while OPT is able to avoid doing the same. However, as the construction employed in the proof of Theorem 4.5 suggests, if an instance of  $LSP_B$  is difficult for  $G$ , then  $C$  is so large that no demand in  $A$  can be driven nearly to zero by  $G$  but not by OPT.

Finally, as an interesting problem generalization, suppose that every user's demand must be greater than or equal to some minimum threshold value  $\tau \in [0, C]$ . This models a system in which each resource has some fixed capacity  $\tau < C$  that is dedicated to the primary user. (For example, in a distributed computing realization, a user's operating system kernel must remain resident in his local memory.) Let  $LSP_\tau$  denote all instances of this problem.

**Corollary 4.6.**  $LSP_\tau$  is NP-complete.

**Proof.** Follows immediately from Theorem 2.2 ( $LSP_\tau$  includes  $LSP_0$  as a special case).  $\square$

**Theorem 4.7.** *For  $LSP_\tau$ ,  $(2C+4\tau)/(C+5\tau) \leq R_G \leq 2C/(C+\tau)$ . Moreover, these bounds are tight when  $\tau=0$  or  $\tau=C$ .*

**Proof.** Suppose  $\tau \in (0, C)$ . To demonstrate the lower bound, modify Example 3.1 as follows. Add  $\tau$  to every demand, and hence to  $C$  as well. Thus  $C=\tau+k+1$  gives both  $C$  and  $\tau$  in terms of  $k$ . (For instance, if  $\tau=C/2$ , then  $C=2k+2$  and  $\tau=k+1$ .)

Therefore  $\text{OPT}(I_k) = \text{OPT}_A(I_k) + \text{OPT}_B(I_k) = kC + ((2k-1)\tau + k) = kC + 2k\tau + k - \tau$  and  $G(I_k) = G_A(I_k) + G_B(I_k) = k\tau + k(C-\tau)/2 + k + ((2k-1)\tau + k) = k(C+\tau)/2 + 2k\tau + 2k - \tau$ . So  $R_G \geq (2C+4\tau+(2-2\tau/k))/(C+5\tau+(4-2\tau/k))$ , which approaches arbitrarily close to  $(2C+4\tau)/(C+5\tau)$  from below as  $k$  grows without bound (since, for any fixed  $\tau$ ,  $\tau/k$  is bounded above by a constant).

To prove the upper bound, let  $I$  denote any instance of  $\text{LSP}_\tau$ . Clearly,  $\text{OPT}_A(I) \leq |A^{(0)}|C$ . From the proof of the chain lemma and its application in Theorem 3.4, we derive  $G_A(I) > |A^{(0)}|(\tau + (C-\tau)/2) = |A^{(0)}|(C+\tau)/2$ . Therefore, since  $\text{OPT}_B(I) = G_B(I)$ , we have

$$\frac{\text{OPT}(I)}{G(I)} < \frac{|A^{(0)}|C + \text{OPT}_B(I)}{|A^{(0)}|(C+\tau)/2 + G_B(I)} \leq \frac{|A^{(0)}|C}{|A^{(0)}|(C+\tau)/2} = \frac{2C}{C+\tau}.$$

If  $\tau=0$ , then the range of values specified in Theorem 4.7 collapses to 2, which is confirmed by Corollary 3.5. If  $\tau=C$ , then the range of values collapses to 1, which is confirmed by observing that no sharing exists in this extreme case.  $\square$

## 5. Directions for future research

Several interesting questions remain unanswered. Is there any polynomial-time algorithm  $\text{ALG}$  with  $R_{\text{ALG}} < R_G$  for  $\text{LSP}$ ? What is the exact value of  $R_G$  for  $\text{LSP}_B$ ? Can our bounds on  $R_G$  for  $\text{LSP}_\tau$  be tightened when the threshold  $\tau$  is strictly greater than zero and less than  $C$ ? (For example, Theorem 4.7 only guarantees that  $8/7 \leq R_G \leq 4/3$  when  $\tau=C/2$ .) And what of other special cases, such as when  $|A^{(0)}| \geq |B^{(0)}|$  or when there is an upper bound on any demand (say, for example,  $2C$ )?

The study of limited sharing can be expanded in a number of ways. One might permit  $k$  users to share another's excess for some fixed  $k > 1$ . The requirement that total system demand saturates total resource capacity could be eased (although this may no longer model resource balancing, an interpretation mentioned in Section 1). Lest the reader be left with any suspicion that  $\text{LSP}$ , as defined here, is overly constrained, we close with the following result. Let us remove the restriction that every user must possess the same resource capacity. Let  $C_i$  denote the (now arbitrary) capacity of user  $i$  and let  $\text{LSP}_C$  denote this relaxed version of  $\text{LSP}$ .

**Theorem 5.1.** *It is NP-complete to determine whether an arbitrary instance of  $\text{LSP}_C$  has any feasible solution.*

**Proof.** Modify the proof of Theorem 2.2 (in which  $Q$  is now insignificant), by setting  $l_1 = l_2 = s/2$  with  $C_1 = C_2 = 0$ , and setting  $l_{i+2} = 0$  with  $C_{i+2} = p_i$  for  $1 \leq i \leq m$ . This instance of  $\text{LSP}_C$  has a feasible solution if and only if the answer to the given instance of  $\text{PARTITION}$  is "yes".  $\square$

## Acknowledgement

We wish to thank the two anonymous referees, whose careful review of our original submission helped to streamline and clarify the presentation of these results.

## References

- [1] M.R. Fellows, D.K. Friesen and M.A. Langston, On finding optimal and near-optimal lineal spanning trees, *Algorithmica* 3 (1988) 549–560.
- [2] D.K. Friesen and M.A. Langston, Variable sized bin packing, *SIAM J. Comput.* 15 (1986) 222–230.
- [3] D.K. Friesen and M.A. Langston, Analysis of a compound bin packing algorithm (to appear).
- [4] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness* (Freeman, New York, 1979).
- [5] D.E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms* (Addison-Wesley, Reading, MA, 1969).
- [6] M.A. Langston, Interstage transportation planning in the deterministic flow-shop environment, *Oper. Res.* 35 (1987) 556–564.
- [7] M.A. Langston and M.P. Morford, Resource allocation under limited sharing, *Computer Science Tech. Rept. CS-87-164*, Washington State University, Pullman, WA (1987).
- [8] C.H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity* (Prentice-Hall, Englewood Cliffs, NJ, 1982).
- [9] W.W. Tsang, Analysis of the square-the-histogram method for generating discrete random variables, M.S. Thesis, Department of Computer Science, Washington State University, Pullman, WA (1980).
- [10] W.W. Tsang and G. Marsaglia, A decision tree algorithm for squaring the histogram in random number generation, in: *Proceedings Australia-Singapore Joint Conference on Information Processing and Combinatorial Mathematics*, Singapore (1986) 325–336.

Reprinted from

# DISCRETE MATHEMATICS

---

Discrete Mathematics 182 (1998) 191–196

On algorithmic applications of the immersion order<sup>1</sup>

An overview of ongoing work presented at the Third Slovenian  
International Conference on Graph Theory

Michael A. Langston, Barbara C. Plaut\*

*Department of Computer Science, University of Tennessee, Knoxville, TN 37996-1301, USA*

Received 18 September 1995; received in revised form 23 November 1996; accepted 15 May 1997



ELSEVIER

## DISCRETE MATHEMATICS

**Editor-in-Chief** Peter L. Hammer, Piscataway (NJ)

### Advisory Editors

C. Berge, Paris  
A.J. Hoffman,  
Yorktown Heights (NY)

V.L. Klee, Seattle (WA)  
R.C. Mullin, Waterloo  
G.-C. Rota, Cambridge (MA)

V.T. Sós, Budapest  
J.H. van Lint, Eindhoven

### Board of Editors

M.S. Aigner, Berlin  
B. Alspach, Burnaby  
G.E. Andrews, Univ. Park (PA)  
A. Barlotti, Firenze  
C. Benzaken, Grenoble  
J.-C. Bermond,  
Sophia-Antipolis  
N.L. Biggs, London  
B. Bollobás, Memphis (TN)  
R.A. Brualdi, Madison (WI)  
T.H. Brylawski,  
Chapel Hill (NC)  
P.J. Cameron, London  
P. Camion, Le Chesnay  
G. Chartrand, Kalamazoo (MI)

V. Chvátal, Piscataway (NJ)  
D. Foata, Strasbourg  
A.S. Fraenkel, Rehovot  
P. Frankl, Tokyo  
A.M. Frieze, Pittsburgh (PA)  
I.M. Gessel, Waltham (MA)  
R.L. Graham,  
Florham Park (NJ)  
A. Hajnal, Budapest  
F. Harary, Las Cruces (NM)  
D.M. Jackson, Waterloo  
J. Kahn, Piscataway (NJ)  
G.O.H. Katona, Budapest  
D.J. Kleitman,  
Cambridge (MA)

A.V. Kostochka, Novosibirsk  
L. Lovász, New Haven (CT)  
I. Rival, Ottawa  
A. Rosa, Hamilton  
S. Rudeanu, Bucharest  
H. Sachs, Ilmenau  
J. Schonheim, Tel-Aviv  
N.J.A. Sloane  
Florham Park (NJ)  
C. Thomassen, Lyngby  
W.T. Tutte, Newmarket  
D.J.A. Welsh, Oxford  
R. Wille, Darmstadt  
D.R. Woodall, Nottingham  
H.P. Yap, Singapore

**Editorial Manager** Nelly Segal **Issue Manager** Mick van Gijlswijk

**Publication Information.** Discrete Mathematics (ISSN 0012-365X). For 1998 volumes 178–193 are scheduled for publication. A combined subscription to Discrete Mathematics and Discrete Applied Mathematics (Vols. 80–88) at reduced rate is available. Subscription prices are available upon request from the Publisher. Subscriptions are accepted on a prepaid basis only and are entered on a calendar year basis. Issues are sent by surface mail except to the following countries where air delivery via SAL is ensured: Argentina, Australia, Brazil, Canada, Hong Kong, India, Israel, Japan, Malaysia, Mexico, New Zealand, Pakistan, China, Singapore, South Africa, South Korea, Taiwan, Thailand, USA. For all other countries airmail rates are available upon request. Claims for missing issues must be made within six months of our publication (mailing) date. For orders, claims, product enquiries (no manuscript enquiries) please contact the Customer Support Department at the Regional Sales Office nearest to you:

**New York,** Elsevier Science, P.O. Box 945, New York, NY 10159-0945, USA. Tel: (+1) 212-633-3730, [Toll Free number for North American Customers: 1-888-4ES-INFO (437-4636)], Fax: (+1) 212-633-3680, E-mail: [usinfo-f@elsevier.com](mailto:usinfo-f@elsevier.com)

**Amsterdam,** Elsevier Science, P.O. Box 211, 1000 AE Amsterdam, Netherlands, Tel: (+31) 20-485-3757, Fax: (+31) 20-485-3432, E-mail: [nlinfo-f@elsevier.nl](mailto:nlinfo-f@elsevier.nl)

**Tokyo,** Elsevier Science, 9-15, Higashi-Azabu 1-chome, Minato-ku, Tokyo 106, Japan. Tel: (+81) 3-5561-5033, Fax: (+81) 3-5561-5047, E-mail: [info@elsevier.co.jp](mailto:info@elsevier.co.jp)

**Singapore,** Elsevier Science, No. 1 Temasek Avenue, # 17-01 Millenia Tower, Singapore 039192. Tel: (+65) 434-3727, Fax: (+65) 337-2230, E-mail: [asiainfo@elsevier.com.sg](mailto:asiainfo@elsevier.com.sg)

© 1998, Elsevier Science B.V. (North-Holland)

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior permission of the Publisher, Elsevier Science B.V., Copyright and Permissions Department, P.O. Box 521, 1000 AM Amsterdam, Netherlands.

**Special regulations for authors**—Upon acceptance of an article by the journal, the author(s) will be asked to transfer copyright of the article to the Publisher. This transfer will ensure the widest possible dissemination of information.

**Special regulations for readers in the USA**—This journal has been registered with the Copyright Clearance Center, Inc. Consent is given for copying of articles for personal or internal use, or for the personal use of specific clients. This consent is given on the condition that the copier pays through the Center the per-copy fee stated in the code on the first page of each article for copying beyond that permitted by Sections 107 or 108 of the US Copyright Law. The appropriate fee should be forwarded with a copy of the first page of the article to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, USA. If no code appears in an article, the author has not given broad consent to copy and permission to copy must be obtained directly from the author. The fee indicated on the first page of an article in this issue will apply retroactively to all articles published in the journal, regardless of the year of publication. This consent does not extend to other kinds of copying such as for general distribution, resale, advertising and promotion purposes, or for creating new collective works. Special written permission must be obtained from the Publisher for such copying.

No responsibility is assumed by the Publisher for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions or ideas contained in the material herein. Although all advertising material is expected to conform to ethical standards, inclusion in this publication does not constitute a guarantee or endorsement of the quality or value of such product or of the claims made of it by its manufacturer.

© The paper used in this publication meets the requirements of ANSI/NISO Z39.48-1992 (Permanence of Paper)

Published monthly

0012-365X/98/\$19.00

Printed in the Netherlands

## On algorithmic applications of the immersion order<sup>1</sup>

An overview of ongoing work presented at the Third Slovenian  
International Conference on Graph Theory

Michael A. Langston, Barbara C. Plaut\*

Department of Computer Science, University of Tennessee, Knoxville, TN 37996-1301, USA

Received 18 September 1995; received in revised form 23 November 1996; accepted 15 May 1997

---

### Abstract

A snapshot of our current exploration of the algorithmic aspects of the immersion order is presented. Integrated circuit partitioning is used as a prototypical applications domain. Decision and search algorithms, self-reductions, closure-preserving operators and related developments are discussed.

---

### 1. Background

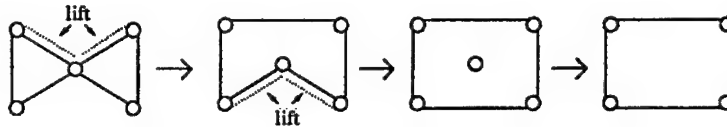
We consider only finite, undirected graphs.  $H$  is said to be *immersed* in  $G$ , written  $H \leq_i G$ , iff a graph isomorphic to  $H$  can be obtained from  $G$  by lifting pairs of adjacent edges and taking a subgraph. A pair of adjacent edges  $uv, vw$ , with  $u \neq v \neq w$  is lifted by removing  $uv$  and  $vw$  and adding  $uw$ . As an example, observe that  $C_4$  is immersed in  $K_1 + 2K_2$  (Fig. 1).

Suppose a family  $F$  is *closed* in this order, that is,  $G \in F$  and  $H \leq_i G \Rightarrow H \in F$ . The *obstruction set* for  $F$  consists of the immersion-minimal elements in  $F$ 's complement. Accordingly,  $F$  has the following characterization:  $G$  is in  $F$  iff no obstruction for  $F$  is immersed in  $G$ . It is known [15] that any such obstruction set is finite. It is also known [7, 14] that deciding whether  $H \leq_i G$  is decidable in polynomial time for every fixed  $H$ . Thus, there exists a polynomial-time recognition algorithm for any immersion-closed family of graphs. See [7] for many examples. Such an algorithm is not constructively known, but possesses a time bound of  $O(n^{h+3})$ , where  $h$  denotes the order of the largest obstruction.

---

\* Corresponding author.

<sup>1</sup> This research is supported in part by the National Science Foundation under grant CDA-9115428 and by the Office of Naval Research under contracts N00014-90-J-1855 and N00014-94-1-1155.

Fig. 1.  $C_4 \leq K_1 + 2K_2$ .

One of the earliest and best-known applications of the immersion order is the *min cut linear arrangement* problem [9]. Though  $\mathcal{NP}$ -complete in general, the fixed-parameter version of this problem has been shown to be decidable in linear time with the aid of the immersion order and special tools based on the treewidth metric [7,2]. Much less is known, however, about the vast majority of applications.

## 2. Circuit partitioning

Consider the field programmable gate array (henceforth FPGA), a collection of logic blocks with programmable connections (see [12]). A given circuit is implemented by partitioning its logic into blocks and connecting the blocks as required (Fig. 2).

Since circuits are frequently too large to fit on a single chip, they must be partitioned over several FPGA's. In building systems with multiple FPGA's, fabrication technology imposes severe restrictions: limits on pin counts (I/O cells) affect inter-chip connectivity; limits on chip area and density bound FPGA sizes.

Such practical limitations motivate many interesting combinatorial problems. Consider, for example, the problem we herewith call the Min Degree Graph Partition problem. In this problem, we are given a graph  $G=(V,E)$  and two integers  $k$  and  $d$ , and are asked whether  $V$  can be partitioned into disjoint subsets  $V_1, V_2, \dots, V_m$  so that, for  $1 \leq i \leq m$ ,  $|V_i| \leq k$  and at most  $d$  edges have exactly one end-point in  $V_i$ . In a multi-FPGA context, for example,  $G$  models the circuit to be partitioned,  $k$  denotes the maximum number of logic blocks permitted on a chip, and  $d$  represents the maximum degree or pin count of any chip.

This problem is clearly very difficult, in fact intractable without parameter bounds, via a reduction from Multiway Cut or Graph Bisection:

**Theorem 1** (Govindan [10]). *Min Degree Graph Partition is  $\mathcal{NP}$ -complete.*

Fortunately, however, the aforementioned fabrication limits can be used to advantage. As long as  $k$  and  $d$  are bounded, the family of 'yes' instances is closed in the immersion order.

**Theorem 2.** *For any fixed  $k$  and  $d$ , Min Degree Graph Partition can be decided in polynomial time.*

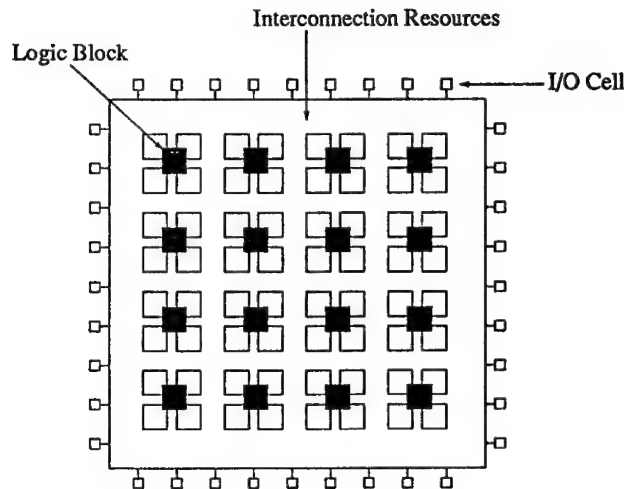


Fig. 2. The FPGA.

**Proof** (sketch). It is straightforward to check that neither taking a subgraph nor lifting pairs of edges can turn a ‘yes’ instance into a ‘no’ instance. Hence, the ‘yes’ family is immersion closed.  $\square$

The last theorem is of particular interest in light of the observation that, unlike Multiway Cut, Min Degree Graph Partition has no known brute-force polynomial-time algorithm when  $k$  and  $d$  are fixed. This is in contrast to the superficially similar Graph Partition problem, in which the cost of a solution is summed over all subsets rather than measured over each, thus bounding the maximum number of partitions.

Results such as this inherently rely on the existence of finite lists of immersion-minimal obstructions. As of this writing, little is known about such obstructions in general or about practical immersion tests in particular. As with the minor order, we expect that even partial sets can be useful [13]. It has been observed that complete graphs are often obstructions to immersion-closed families. Testing for  $K_1$ ,  $K_2$  or  $K_3$  is easy. Testing for  $K_4$  turns out to be quite complicated, however, though achievable in linear time. See [3] for decision, search and parallel algorithms.

Min Degree Graph Partition is an excellent example of the current state of the art. We have identified a wide array of other problems, largely from the circuit partitioning domain, amenable to tools based on the immersion order. For most of these, just as with Min Degree Graph Partition, we can at present say not much more than that they are (nonconstructively) decidable in polynomial time. Whether they are solvable in low-order polynomial time, perhaps even linear time, is an open question, and one we are actively pursuing. One might be tempted to employ the treewidth metric, useful for Min Cut Linear Arrangement. If the family of ‘yes’ instances has bounded treewidth, linear time recognizability is assured. But that is not generally

the case. To see this, consider Min Degree Graph Partition with  $k=1$  and  $d=4$ . Even this simple family of graphs contains the  $w \times w$  grid for any  $w$ , a graph with treewidth  $w$ . One might also ask about eliminating nonconstructivity. We have developed some techniques for that task, although they are mainly of theoretical interest and beyond the scope of this brief review. We refer the reader to [8] for details.

### 3. Search algorithms and self-reducibility

It is sometimes possible to solve a search problem by reducing it to a related decision problem. (See [9] for a detailed discussion of search versus decision.) For example, one might seek to find a satisfying subset assignment for Min Degree Graph Partition with the aid of a routine that merely tells whether such an assignment exists.

This approach to algorithm design is called *self-reducibility*, and has been formulated in many ways in the literature. In its most limited form, an assortment of restrictions are placed on the decision algorithm, its input and the lexicographic position of the output produced (see, e.g., [16]). In more general forms, input/output limitations are eliminated and decision algorithms quite distant from the original problem are permitted (see, e.g., [6]). Additional variations exist, some even incorporating randomness or parallelism (see, e.g., [4, 11]).

It is not difficult to see that, for any fixed  $k$  and  $d$ , Min Degree Graph Partition is self-reducible in polynomial time. That is, one can construct a satisfying subset assignment, if any exist, with at most a polynomial number of calls to a decision algorithm, known from the last section also to run in polynomial time.

It can in fact be self-reduced with only a linear number of calls.

**Theorem 3.** *For any fixed  $k$  and  $d$ , the search version of Min Degree Graph Partition can be solved in  $O(np(n))$  time, where  $p(n)$  denotes the time required to solve the decision version of the problem.*

**Proof** (sketch). No vertex in a ‘yes’ instance has  $d+k$  or more neighbors (a star with  $d+k$  rays is an obstruction). Furthermore, in such an instance, there must exist some satisfying assignment in which each subset induces a connected subgraph. From this it can be shown that, no matter the rest of the partition, two vertices not connected by a sufficiently short path need never share the same subset. Thus we know in advance that, as a solution is recursively constructed, a vertex  $v$  need share a subset only with candidates from a bounded-size neighborhood. Each such candidate,  $u$ , can be tested for suitability by adding  $d+1$  copies of the edge  $uv$ , calling the decision algorithm and retaining the extra edges only when the resulting graph is also a ‘yes’ instance.  $\square$

A number of interesting self-reducibility issues remain open for this order, though none yet are perhaps as noteworthy as embedding reducibilities are for the minor order (where the permitted operations are subgraph and edge contraction). For example, knotlessness [8] is decidable in polynomial time, though it is not known to be searchable within any time complexity class.

#### 4. Closure-preserving operators

In the case of a ‘no’ instance, some sort of approximation scheme [9] is often required. But increasing the size of problem parameters may not be desirable or even possible in many settings. An approach with some practical appeal then is to ask instead whether one can modify the graph (simplify the underlying circuit) so that it becomes a ‘yes’ instance. More generally, we seek systematic methods for making such modifications so as to preserve immersion closure.

Let  $F$  denote a family of graphs, and let  $F_v(h)$  denote those graphs for which there exists some set of  $h$  or fewer vertices whose removal creates a graph in  $F$ . When  $h$  is fixed, recognizing  $F_v(h)$  can of course be reduced to recognizing  $F$  by brute force in time proportional to  $n^h$ , a polynomial. If  $F$  is minor-closed, however, there is a more efficient technique. It is known [5] that if  $F$  is minor-closed, then so is  $F_v(h)$ .

Unfortunately, this operator does not work for the immersion order. To see this, let  $F$  denote the family of edgeless graphs, and let  $h = 1$ . The star graph with three rays is in  $F_v(1)$ , but the graph obtained by lifting a pair of edges yields a matching of size two, which is not in  $F_v(1)$ .

So consider edges instead, and let  $F_e(h)$  denote those graphs for which there exists some set of  $h$  or fewer edges whose removal creates a graph in  $F$ .

**Theorem 4.** *For any fixed  $h$ , if  $F$  is immersion-closed, then so is  $F_e(h)$ .*

This operator, plus self-reducibility, therefore yields a polynomial-time approach for solving the decision and search versions of  $F_e(h)$  when, for example,  $F$  denotes Min Degree Graph Partition. Other operators exist, but this is perhaps the most natural from an algorithmic standpoint.

#### 5. In closing

Much is known about complexity-theoretic issues for subgraph, topological and even minor containment [1]. In contrast, we have thus far really only scratched the surface in understanding some of the range and depth of algorithmic applications of the immersion order. Many challenging open questions beckon, several of which we have attempted to illuminate here.

## References

- [1] D. Bienstock, M.A. Langston, Algorithmic implications of the graph minor theorem, in: M.O. Ball, T.L. Magnanti, C.L. Monma, G.L. Nemhauser (Eds.), *Handbook of Operations Research and Management Science: Network Models*, North-Holland, Amsterdam, 1995, pp. 481–502.
- [2] H.L. Bodlaender, A linear time algorithm for finding tree-decompositions of small treewidth, Technical Report, Utrecht University, 1992.
- [3] H. Booth, R. Govindan, M.A. Langston, S. Ramachandramurthi, Sequential and parallel algorithms for  $K_4$  immersion testing, Technical Report, University of Tennessee, 1995.
- [4] J. Feigenbaum, L. Fortnow, Random self-reducibility of complete sets, *SIAM J. Comput.* 22 (1993) 994–1005.
- [5] M.R. Fellows, M.A. Langston, Nonconstructive tools for proving polynomial-time decidability, *J. ACM* 35 (1988) 727–739.
- [6] M.R. Fellows, M.A. Langston, Fast search algorithms for layout permutation problems, *Internat. J. Comput. Aided VLSI Design* 3 (1991) 325–342.
- [7] M.R. Fellows, M.A. Langston On well-partial-order theory and its application to combinatorial problems of VLSI design, *SIAM J. Discrete Math.* 5 (1992) 117–126.
- [8] M.R. Fellows, M.A. Langston, On search, decision and the efficiency of polynomial-time algorithms, *J. Comput. Systems Sci.* 49 (1994) 769–779.
- [9] M.R. Garey, D.S. Johnson, *Computers and intractability: A guide to the theory of  $\mathcal{NP}$ -completeness*, Freeman, San Francisco, CA, 1979.
- [10] R. Govindan, private communication.
- [11] R.M. Karp, E. Upfal, A. Wigderson, The complexity of parallel search, *J. Comput. Systems Sci.* 36 (1988) 225–253.
- [12] R. Murgai, Y. Nishizaki, N. Shenoy, R.K. Brayton, A. Sangiovanni-Vincentelli, Logic synthesis for programmable gate arrays, *Proc. 27th ACM/IEEE Design Automation Conf.* (1990) 620–625.
- [13] S. Ramachandramurthi, Algorithms for VLSI layout based on graph width metrics, Ph.D. Dissertation, University of Tennessee, 1994.
- [14] N. Robertson, P.D. Seymour, Graph minors XIII. The disjoint paths problem, *J. Combin. Theory Ser. B* 63 (1995) 65–110.
- [15] N. Robertson, P.D. Seymour, Graph minors XXIII. The Nash-Williams immersion conjecture, to appear.
- [16] C.P. Schnorr, Optimal algorithms for self-reducible problems, *Proc. 1976 Internat. Conf. on Automata, Programming and Languages*, 1976, pp. 322–337.

### Scope of the Journal

The aim of this journal is to bring together research papers in different areas of discrete mathematics. Contributions presented to the journal can be research papers, short notes, surveys, and possibly research problems. The 'Communications' section will be devoted to the fastest possible publication of the brief outlines of recent research results, the detailed presentation of which might be submitted for possible publication in DISC or elsewhere. The journal will also publish a limited number of book announcements, as well as proceedings of conferences. The journal will publish papers in combinatorial mathematics and related areas. In particular, graph and hypergraph theory, network theory, coding theory, block designs, lattice theory, the theory of partially ordered sets, combinatorial geometries, matroid theory, extremal set theory, logic and automata, matrices, polyhedra, discrete probability theory, etc. shall be among the fields covered by the journal.

### Instructions to contributors

All contributions should be written in English or French, should have an abstract in English (as well as one in French if the paper is written in French), and—with the exception of Communications—should be sent in triplicate to Nelly Segal, Editorial Manager, RUTCOR, Rutgers University Center for Operations Research, P.O. Box 5062, New Brunswick, NJ 08903-5062, USA. The authors are requested to put their mailing address on the manuscript.

Upon acceptance of an article, the author(s) will be asked to transfer copyright of the article to the Publisher. This transfer will ensure the widest possible dissemination of information.

Manuscripts submitted for the Communications section, having at most 5 typewritten pages, should be sent to a member of the editorial board in triplicate. Detailed proofs do not have to be included, but results must be accompanied at least by rough outlines of their proofs. Subsequent publication in this journal or elsewhere of the full text of a research report, the outline of which has been published in the Communications section of our journal, is not excluded. Every effort shall be made for the fastest possible publication of Communications.

Please make sure that the paper is submitted in its final form. Corrections in the proofstage, other than of printer's errors, should be avoided; costs arising from such extra corrections will be charged to the authors.

The manuscript should be prepared for publication in accordance with instructions given in the 'Instructions to Authors' (available from the Publisher) details of which are condensed below:

1. The manuscript must be typed on one side of the paper in double spacing with wide margins. A duplicate copy should be retained by the author.
2. Special care should be given to the preparation of the drawings for figures and diagrams. Except for a reduction in size, they will appear in the final printing in exactly the same form as they were submitted by the author; normally they will not be redrawn by the printer. In order to make a photographic reproduction possible, all drawings should be on separate sheets, with wide margins, drawn large size, in Indian ink, and carefully lettered. Exceptions are diagrams only containing formulae and a small number of single straight lines (or arrows); these can be typeset by the printer.
3. References should be listed alphabetically, in the same way as the following examples:  
*For a book:* W.K. Chen, *Applied Graph Theory* (North-Holland, Amsterdam, 1971).  
*For a paper in a journal:* M.M.G. Fase and M. van Tol, The monetary return on investment in paintings, *Econom. Statist. Ber.* 79 (1994) 684–689.  
*For a paper in a contributed volume:* M.O. Rabin, Weakly definable relations and special automata, in: Y. Bar-Hillel, ed., *Mathematical Logic and Foundations of Set Theory* (North-Holland, Amsterdam, 1970) 1–23.  
*For an unpublished paper:* R. Schrauwen, *Series of singularities and their topology*, Ph.D. Thesis, Utrecht University, Utrecht, 1991.

### Instructions for LaTeX manuscripts

The LaTeX files of papers that have been accepted for publication may be sent to the Publisher by e-mail or on a diskette (3.5" or 5.25" MS-DOS). If the file is suitable, proofs will be produced without rekeying the text. The article should be encoded in Elsevier-LaTeX, standard LaTeX, or AMS-LaTeX (in document style "article"). The Elsevier-LaTeX package, together with instructions on how to prepare a file, is available from the Publisher. This package can also be obtained through the Elsevier WWW home page (<http://www.elsevier.nl/>), or using anonymous FTP from the Comprehensive TeX Archive Network (CTAN). The host-names are: ftp.dante.de, ftp.tex.ac.uk, ftp.shsu.edu; the CTAN directory is: /tex-archive/macros/latex/contrib/supported/elsevier. *No changes from the accepted version are permissible, without the explicit approval by the Editor. The Publisher reserves the right to decide whether to use the author's file or not.* If the file is sent by e-mail, the name of the journal *Discrete Mathematics*, should be mentioned in the "subject field" of the message to identify the paper. Authors should include an ASCII table (available from the Publisher) in their files to enable the detection of transmission errors. The files should be mailed to: Ms. Paulette de Boer, Elsevier Science B.V., P.O. Box 103, 1000 AC Amsterdam, Netherlands, Fax: (31-20) 4852616. E-mail: [p.boer@elsevier.nl](mailto:p.boer@elsevier.nl).

### Author's benefits

1. 30% discount on all book publications of North-Holland.
2. 50 reprints are provided free of charge to the principal author of each paper published.

*US mailing notice*—*Discrete Mathematics* (0012-365x) is published (total 16 issues) by Elsevier Science (Molenwerf 1, Postbus 211, 1000 AE Amsterdam). Annual subscription price in the USA US\$ 3437.00 (US\$ price valid in North, Central and South America only), including air speed delivery. Application to mail at periodicals postage rate is pending at Jamaica, NY 11431.

USA POSTMASTERS: Send address changes to Discrete Mathematics, Publication Expediting, Inc., 200 Meacham Avenue, Elmont, NY 11003. Air freight and mailing in the USA by Publication Expediting.

Reprinted from

# Information Processing Letters

---

Information Processing Letters 68 (1998) 17–23

## Approximating the pathwidth of outerplanar graphs<sup>☆</sup>

Rajeev Govindan<sup>a</sup>, Michael A. Langston<sup>b,\*</sup>, Xudong Yan<sup>c</sup>

<sup>a</sup> *Qualcomm Incorporated, San Diego, CA 92121, USA*

<sup>b</sup> *Department of Computer Science, University of Tennessee, Knoxville, TN 37996, USA*

<sup>c</sup> *Prism Solutions Inc, Sunnyvale, CA 94089, USA*

Received 22 October 1997; received in revised form 4 April 1998

Communicated by S.E. Hambrusch



## Board of Editors

- S.G. Akl, *Department of Computing and Information Science, Queen's University, Kingston, Canada K7L 3N6* akl@qucis.queensu.ca  
G.R. Andrews, *Computer Science Department, University of Arizona, Tucson, AZ 85721, USA* greg@cs.arizona.edu  
L. Boasson, *Laboratoire Informatique Théorique et Programmation (LITP), UFR d'Informatique, Université Paris 7, 2 Place Jussieu, 75251 Paris Cédex 05, France* Luc.Boasson@liafa.jussieu.fr  
F.Y.L. Chin, *The University of Hong Kong, Department of Computer Science, Pokfulam Road, Hong Kong* chin@cs.hku.hk  
F. Dehne, *School of Computer Science, Carleton University, Ottawa, Canada K1S 5B6* dehne@scs.carleton.ca  
D. Dolev, *Institute of Computer Science, Hebrew University of Jerusalem, Givat Ram, 91904 Jerusalem, Israel* dolev@cs.huji.ac.il  
R.G. Dromey, *School of Computing and Information Technology, Griffith University, Nathan Campus, Kessels Road, Brisbane, QLD 4111, Australia* rgd@cit.gu.edu.au  
J.L. Fiadeiro, *Universidade de Lisboa, Faculdade de Ciencias, Departamento de Informática, Bioco C5 – Campo Grande, 1700 Lisboa, Portugal* llf@di.fc.ul.pt  
H. Ganzinger, *Max-Planck-Institut für Informatik, Im Stadtwald – Gebäude 46.1, D-66123 Saarbrücken, Germany* hg@mpi-sb.mpg.de  
D. Gries, *Department of Computer Science, Cornell University, 4115 Upson Hall, Ithaca, NY 14853-7501, USA* gries@cs.cornell.edu  
S.E. Hambrusch, *216 Computer Science Building, Department of Computer Sciences, Purdue University, West Lafayette, IN 47907, USA* seh@cs.purdue.edu  
L.A. Hemaspaandra, *Department of Computer Science, University of Rochester, Rochester, NY 14627, USA* lane@cs.rochester.edu  
K. Iwama, *Kyoto University, Department of Information Science, Sakyo, Kyoto 606-01, Japan* iwama@kuis.kyoto-u.ac.jp  
T. Lengauer, *GMD-11, Schloss Birlinghoven, D-53757 Sankt Augustin, Germany* lengauer@gmd.de  
C. Morgan, *Oxford University Computing Laboratory, Programming Research Group, Wolfson Building, Parks Road, Oxford OX1 3QD, United Kingdom* carroll@prg.oxford.ac.uk  
F.B. Schneider, *Department of Computer Science, Cornell University, 4130 Upson Hall, Ithaca, NY 14853-7501, USA* fbs@cs.cornell.edu  
A. Tarlecki, *Institute of Informatics, Warsaw University, ul. Banacha 2, 02-097 Warsaw, Poland* tarlecki@mimuw.edu.pl  
W.M. Turski, *Institute of Informatics, Warsaw University, ul. Banacha 2, 02-097 Warsaw, Poland* wmt@mimuw.edu.pl  
P.M.B. Vitányi, *Centre for Mathematics and Computer Science (CWI), Kruislaan 413, 1098 SJ Amsterdam, The Netherlands* paul.vitanyi@cw.nl  
S. Zaks, *Department of Computer Science, Technion – Israel Institute of Technology, Technion City, Haifa 32000, Israel* zaks@cs.technion.ac.il

The Managing Editors of *IPL* are F. Dehne (dehne@scs.carleton.ca), D. Gries (gries@cs.cornell.edu), and W.M. Turski (wmt@mimuw.edu.pl). Please contact one of them if you have questions or problems concerning submission, processing, or publication of a paper. However, the Managing Editors are not responsible for directing submissions to appropriate editors. Therefore, when submitting a paper, select an Editor for reasons of geographic or subject-area proximity (see Instructions for Authors on the inside back cover).

## Publication Information

Information Processing Letters (ISSN 0020-0190). For 1998 volumes 65–68 are scheduled for publication.

Subscription prices are available upon request from the Publisher.

Subscriptions are accepted on a prepaid basis only and are entered on a calendar year basis. Issues are sent by surface mail except to the following countries where air delivery via SAL is ensured: Argentina, Australia, Brazil, Canada, Hong Kong, India, Israel, Japan, Malaysia, Mexico, New Zealand, Pakistan, PR China, Singapore, South Africa, South Korea, Taiwan, Thailand, USA. For all other countries airmail rates are available upon request.

Claims for missing issues must be made within six months of our publication (mailing) date.

For orders, claims, product enquiries (no manuscript enquiries) please contact the Customer Support Department at the Regional Sales Office nearest to you:

**New York**, Elsevier Science, P.O. Box 945, New York, NY 10159-0945, USA. Tel: (+1) 212-633-3730, [Toll free number for North American Customers: 1-888-4ES-INFO (437-4636)], Fax: (+1) 212-633-3680, E-mail: usinfo-f@elsevier.com

**Amsterdam**, Elsevier Science, P.O. Box 211, 1000 AE Amsterdam, The Netherlands. Tel: (+31) 20-485-3757, Fax: (+31) 20-485-3432, E-mail: nlinfo-f@elsevier.nl

**Tokyo**, Elsevier Science, 9-15, Higashi-Azabu 1-chome, Minato-ku, Tokyo 106, Japan. Tel: (+81) 3-5561-5033, Fax: (+81) 3-5561-5047, E-mail: info@elsevier.co.jp

**Singapore**, Elsevier Science, No. 1 Temasek Avenue, #17-01 Millenia Tower, Singapore 039192. Tel: (+65) 434-3727, Fax: (+65) 337-2230, E-mail: asiainfo@elsevier.com.sg

**Rio de Janeiro**, Elsevier Science, Rua Sete de Setembro 111/16 Andar, 20050-002 Centro, Rio de Janeiro – RJ, Brazil. Tel: (+55) (21) 509 5340, Fax: (+55) (21) 507 1991, E-mail: elsevier@campus.com.br [Note (Latin America): for orders, claims and help desk information, please contact the Regional Sales Office in New York as listed above]

US mailing notice – Information Processing Letters (0020-0190) is published semi-monthly by Elsevier Science B.V. (Molenwerf 1, Postbus 211, 1000 AE Amsterdam). Annual subscription price in the USA US\$ 1242.00 (US\$ price valid in North, Central and South America only), including air speed delivery. Application to mail at periodicals postage rate is pending at Jamaica, NY 11431.

USA POSTMASTERS: Send address changes to Information Processing Letters, Publication Expediting, Inc., 200 Meacham Avenue, Elmont, NY 11003. Air freight and mailing in the USA by Publication Expediting.

⊗ The paper used in this publication meets the requirements of ANSI/NISO Z39.48-1992 (Permanence of Paper).

Published semi-monthly

Printed in The Netherlands

## Approximating the pathwidth of outerplanar graphs<sup>☆</sup>

Rajeev Govindan<sup>a</sup>, Michael A. Langston<sup>b,\*</sup>, Xudong Yan<sup>c</sup>

<sup>a</sup> *Qualcomm Incorporated, San Diego, CA 92121, USA*

<sup>b</sup> *Department of Computer Science, University of Tennessee, Knoxville, TN 37996, USA*

<sup>c</sup> *Prism Solutions Inc, Sunnyvale, CA 94089, USA*

Received 22 October 1997; received in revised form 4 April 1998

Communicated by S.E. Hambrusch

---

### Abstract

Pathwidth is a well-known NP-complete graph metric. We present a technique to approximate the pathwidth of outerplanar graphs. Although a polynomial-time algorithm is already known to determine the pathwidth of outerplanar graphs, this algorithm is not practical. Our algorithm works in  $O(n \log n)$  time on graphs of order  $n$ , is practical and produces solutions at most three times the optimum. © 1998 Elsevier Science B.V. All rights reserved.

**Keywords:** Algorithms; Pathwidth; Outerplanar graphs; Approximation; Tree decomposition

---

### 1. Introduction

Pathwidth was defined by Robertson and Seymour in their seminal series of papers on Graph Minors [11]. Since then, this metric has found application in many areas, ranging from circuit layout to natural language processing [6,10]. Determining pathwidth is NP-complete [8]. Thus, it is natural to search for fast approximation algorithms. No polynomial-time relative approximation algorithm (one whose solution is within a multiplicative constant of the optimum) is known for the general problem. Moreover, no polynomial-time absolute approximation algorithm (one whose solution is within an additive constant of the optimum) can exist unless  $P = NP$  [3].

The main result of this paper is a practical relative approximation algorithm for the pathwidth problem

on outerplanar graphs. Since outerplanar graphs have treewidth two or less, the methods in [1] can, in principle, be used to compute the pathwidth exactly in polynomial time. This is not a realistic option, however, because of the high degree of the polynomial (more than four times the treewidth). In contrast, our algorithm approximates the pathwidth to within a factor of three of the optimum in  $O(n \log n)$  time.

### 2. Our approach

We consider only connected graphs without loops or multiple edges.<sup>1</sup>

#### 2.1. Tree and path decompositions

A *tree decomposition* of a graph  $G$  is a pair  $(T, Y)$ , where  $T$  is a tree and  $Y = \{Y_i \mid i \in V(T)\}$  is a

---

<sup>☆</sup> This research is supported in part by the Office of Naval Research under contract N00014-90-J-1855.

\* Corresponding author. Email: langston@cs.utk.edu.

<sup>1</sup> Thus an edge is uniquely specified by its endpoints. For example,  $ab$  denotes an edge between vertex  $a$  and vertex  $b$ .

collection of subsets of  $V(G)$  such that (i) for each edge  $e \in E(G)$ , some  $Y_i$  contains both endpoints of  $e$ , and (ii) for all  $i, j, k \in V(T)$ , if  $j$  is on the path between  $i$  and  $k$  in  $T$ ,  $Y_i \cap Y_k \subseteq Y_j$ . The *width* of a tree decomposition  $(T, Y)$  is one less than the size of the largest set in  $Y$ . The *treewidth* of  $G$  (denoted  $tw(G)$ ) is the smallest width of all its tree decompositions.

A *path decomposition* of  $G$  is a sequence  $X_1, \dots, X_r$  of subsets of  $V(G)$  such that (i) for each edge  $e \in E(G)$ , some  $X_i$  contains both endpoints of  $e$ , and (ii) for  $1 \leq i \leq j \leq k \leq r$ ,  $X_i \cap X_k \subseteq X_j$ . The *width* of a path decomposition  $X_1, \dots, X_r$  is one less than the size of the largest set  $X_i$ ,  $1 \leq i \leq r$ . The *pathwidth* of  $G$  (denoted  $pw(G)$ ) is the smallest width of all its path decompositions.

## 2.2. A conversion procedure

Path decompositions can be derived from tree decompositions. We employ such a procedure, **td2pd**, and prove its correctness. It requires a routine to construct optimal path decompositions of trees. For this, we first construct a layout that minimizes vertex separation using the method presented in [5], and then convert this layout into a path decomposition as described in [9].

### Procedure **td2pd**

**Input:** A tree decomposition  $(T, Y)$  of a graph  $G$ .

**Output:** A path decomposition of  $G$ .

**begin procedure**

$X_1, \dots, X_r :=$  an optimal path decomposition of  $T$ ;

**for**  $1 \leq i \leq r$  **do**

$P_i := \bigcup_{j \in X_i} Y_j$ ;

**output**  $P_1, \dots, P_r$ ;

**end procedure**

Let  $n$  denote the order of  $T$ . Using the results of [5,9], it takes  $O(n \log n)$  time to construct an optimal path decomposition  $X_1, \dots, X_r$  of  $T$ . The following properties hold:  $r \leq n$ , and for  $1 \leq i \leq r$ ,  $|X_i|$  is  $O(\log n)$ . Thus **td2pd** has  $O(n \log n)$  time complexity as long as the input tree decomposition has bounded width. This is true for outerplanar graphs, which have treewidth at most two.

**Theorem 1.** Let  $(T, Y)$  denote a width- $t$  tree decomposition of a graph  $G$ . Then **td2pd** $((T, Y))$  returns

a path decomposition of  $G$  with width no more than  $(t+1)(pw(T)+1)-1$ .

**Proof.** Let  $X_1, \dots, X_r$  denote the optimal path decomposition of  $T$  constructed in **td2pd**, and let  $P_1, \dots, P_r$  denote the output of **td2pd**. Then, for  $1 \leq i \leq r$ ,

$$|P_i| = \left| \bigcup_{j \in X_i} Y_j \right| \leq (t+1)(pw(T)+1).$$

Thus the width condition is satisfied, and we only need to check that  $P_1, \dots, P_r$  is a valid path decomposition of  $G$ .

It is easy to see that  $P_1, \dots, P_r$  covers all edges in  $G$ . We prove by contradiction that  $P_1, \dots, P_r$  has the intersection property. If the intersection property does not hold, then for some  $1 \leq i < j < k \leq r$ , there is a vertex  $v$  in  $P_i \cap P_k$  that is not in  $P_j$ . Since  $v \in P_i \cap P_k$ , there must exist  $l \in X_i$  and  $m \in X_k$ , such that  $v$  belongs to  $Y_l$  and  $Y_m$ . Consider the subsets  $V_1$  and  $V_2$  of  $V(T)$ , where

$$V_1 = \bigcup_{p < j} X_p - X_j \quad \text{and} \quad V_2 = \bigcup_{p > j} X_p - X_j.$$

The intersection property of  $X_1, \dots, X_r$  implies that  $V_1$  and  $V_2$  are disjoint. Moreover, there is no edge in  $T$  connecting  $V_1$  and  $V_2$ , because some  $X_q$  must contain both endpoints of such an edge, contradicting the disjointness of  $V_1$  and  $V_2$ . Thus every path between  $V_1$  and  $V_2$  in  $T$  contains a vertex from  $X_j$ . In particular, the path between  $l$  and  $m$  must contain a vertex, say  $h$ , from  $X_j$ . By the intersection property of  $(T, Y)$ ,  $v \in Y_h$ . Since  $h \in X_j$ ,  $Y_h \subseteq P_j$  and  $v \in P_j$ , a contradiction.  $\square$

## 3. Path decompositions of outerplanar graphs

A graph is *outerplanar* if it has a planar embedding with all vertices lying in a single face. Outerplanar graphs have treewidth at most two. In this section, we develop an algorithm that, for an outerplanar graph  $G$ , constructs an optimal tree decomposition  $(T, Y)$  with  $pw(T) \leq pw(G)$ . By Theorem 1, running **td2pd** on  $(T, Y)$  produces a path decomposition with width at most  $3 \times pw(G) + 2$ .

We say that a tree decomposition  $(T, Y)$  is *simple* if  $(T, Y)$  has width at most two,  $T$  is a subgraph

of  $G$ , and  $v \in Y_v$  for all  $v \in V(T)$ . Because the pathwidth of a subgraph of  $G$  cannot be greater than the pathwidth of  $G$ , if  $(T, Y)$  is simple, then  $\text{pw}(T) \leq \text{pw}(G)$ . Our algorithm constructs  $(T, Y)$  by combining tree decompositions of  $G$ 's subgraphs as described in the following lemma.

**Lemma 2.** Let  $(T', Y')$  and  $(T'', Y'')$  denote tree decompositions of graphs  $G'$  and  $G''$ , respectively. Suppose  $V(T')$  and  $V(T'')$  are disjoint, and there are vertices  $u \in V(T')$  and  $v \in V(T'')$  such that all vertices in  $V(G') \cap V(G'')$  are in both  $Y'_u$  and  $Y''_v$ . Then we may obtain a tree decomposition  $(T, Y)$  of  $G = G' \cup G''$  by setting  $T = T' \cup T'' \cup \{uv\}$  and  $Y = Y' \cup Y''$ .<sup>2</sup> Moreover,  $(T, Y)$  is simple if  $(T', Y')$  and  $(T'', Y'')$  are simple and  $uv \in E(G)$ .

**Proof.** Since  $(T', Y')$  covers all edges in  $G'$  and  $(T'', Y'')$  covers all edges in  $G''$ ,  $(T, Y)$  covers all edges in  $G$ . To verify that  $(T, Y)$  has the intersection property, let  $i, j$  and  $k$  be vertices in  $T$  such that  $j$  is on the path between  $i$  and  $k$ . We need to show that  $Y_i \cap Y_k \subseteq Y_j$ . This follows immediately if both  $i$  and  $k$  belong to  $T'$  or both belong to  $T''$ . So assume, without loss of generality, that  $i \in V(T')$  and  $k \in V(T'')$ . Note that the path between  $i$  and  $k$  contains both  $u$  and  $v$ . If  $j \in V(T')$ , then by the intersection property of  $(T', Y')$ ,

$$Y_i \cap Y_k \subseteq Y_i \cap Y_u = Y'_i \cap Y'_u \subseteq Y'_j = Y_j.$$

If  $j \in V(T'')$ , then by the intersection property of  $(T'', Y'')$ ,

$$Y_i \cap Y_k \subseteq Y_v \cap Y_k = Y''_v \cap Y''_k \subseteq Y''_j = Y_j. \quad \square$$

### 3.1. Biconnected outerplanar graphs

We concentrate initially on biconnected graphs (those without cut points).

**Lemma 3.** Let  $G$  be biconnected, outerplanar and of order at least three. Let  $v$  denote an arbitrary vertex in

$G$ . Then  $G$  contains a path<sup>3</sup>  $P$  with at least two edges such that the following conditions hold:

- all vertices in  $P$ , except possibly its endpoints, have degree two in  $G$ ,
- the endpoints of  $P$  are adjacent in  $G$ , and
- $v$  is either an endpoint of  $P$  or not in  $P$ .

**Proof.** Fix an outerplanar layout of  $G$ . Since  $G$  is biconnected, the outer face of this layout defines a Hamiltonian circuit in  $G$ . Let  $I$  denote the set of internal edges of  $G$  (those not on the external face). The proof proceeds by induction on  $|I|$ . For the basis case,  $|I| = 0$ ,  $G$  is a cycle and the lemma is satisfied by setting  $P$  to  $G - \{uv\}$ , where  $u$  is an arbitrary vertex adjacent to  $v$ . For the induction hypothesis, assume the lemma for  $|I| = i \geq 0$ . For the induction step, consider the case in which  $|I| = i + 1$ . Then  $G$  can be expressed as the union of two smaller biconnected outerplanar graphs,  $G'$  and  $G''$ , each of order at least three, that share only two vertices,  $a$  and  $b$ , and the edge  $ab$ . Assume, without loss of generality, that  $v \in V(G'')$  and that  $v \neq b$ . By hypothesis,  $G'$  contains a path,  $P'$ , all of whose non-endpoint vertices have degree two, whose endpoints are adjacent, and that excludes  $v' = a$  as a non-endpoint vertex. Since  $v \notin V(G') - \{a\}$ ,  $P'$  cannot contain  $v$  as a non-endpoint vertex. Thus setting  $P = P'$  satisfies the lemma.  $\square$

Fig. 1 illustrates Lemma 3, with vertex 9 playing the role of  $v$ . Both paths marked with dashed edges satisfy the lemma. (The path 7-8-9-10 does not satisfy the lemma, because it contains  $v$  as a non-endpoint vertex.)

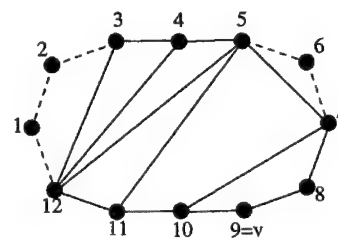
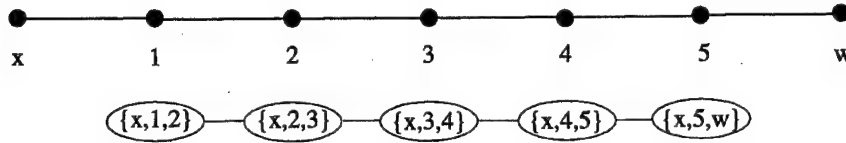


Fig. 1. Paths satisfying Lemma 3.

<sup>2</sup>  $T' \cup T'' \cup \{uv\}$  denotes the tree with vertex set  $V(T') \cup V(T'')$  and edge set  $E(T') \cup E(T'') \cup \{uv\}$ .  $Y = Y' \cup Y''$  denotes the collection  $\{Y_v \mid v \in V(T') \cup V(T'')\}$ , where  $Y_v$  equals either  $Y'_v$  or  $Y''_v$  depending on whether  $v \in V(T')$  or  $v \in V(T'')$ .

<sup>3</sup> If  $v_1, \dots, v_k$  is a sequence of distinct vertices in  $G$  such that  $v_i$  is adjacent to  $v_{i+1}$  for  $1 \leq i < k$ , then the subgraph  $H$  of  $G$  with  $V(H) = \{v_1, \dots, v_k\}$  and  $E(H) = \{v_i v_{i+1} \mid 1 \leq i < k\}$  is said to be a path in  $G$ .

Fig. 2. A path and its  $w$ -extensible tree decomposition.

If a path  $P$  contains at least two edges and has endpoints  $w$  and  $x$ , then it has a width-two tree decomposition  $(T, Y)$  such that  $T = P - \{w, x\}$  and for  $i \in V(T)$ ,  $Y_i = \{x, i, j\}$ , where  $j$  is the neighbor of  $i$  on  $w$ 's side (the sets  $Y_i$  actually form a path decomposition of  $P$ ). We call  $(T, Y)$  a  $w$ -extensible tree decomposition of  $P$ . Fig. 2 shows a path and its  $w$ -extensible tree decomposition. The sets  $Y_i$  are shown inside the ovals.

Note that for every edge  $ij \in E(P)$ , either  $\{i, j\} \subseteq Y_i$  or  $\{i, j\} \subseteq Y_j$ . We use the notion of extensibility to derive **bc-op-td**, our algorithm<sup>4</sup> to construct simple tree decompositions of biconnected outerplanar graphs.

#### Procedure bc-op-td

**Input:** A biconnected outerplanar graph  $G$  of order two or more, and a vertex  $v$  in  $G$ .

**Output:** A simple tree decomposition  $(T, Y)$  of  $G$ , with  $T$  spanning  $G - \{v\}$ .

**begin procedure**

  if  $|V(G)| = 2$

**then begin**

$u :=$  the vertex adjacent to  $v$ ;

$Y_u := \{u, v\}$ ,  $T := \{u\}$ ,  $Y := \{Y_u\}$ ;

**end;**

**else begin**

$P :=$  a path, between some two vertices  $w$  and  $x$ , that satisfies Lemma 3;

$(T', Y') := \text{bc-op-td}(G - (P - \{w, x\}), v)$ ;

**if**  $\{w, x\} \subseteq Y'_w$

**then begin**

$e :=$  the edge incident on  $w$  in  $P$ ;

$(T'', Y'') :=$  the  $w$ -extensible tree decomposition of  $P$ ;

**end;**

**else begin**

$e :=$  the edge incident on  $x$  in  $P$ ;

$(T'', Y'') :=$  the  $x$ -extensible tree decomposition of  $P$ ;

**end;**

$T := T' \cup T'' \cup \{e\}$ ,  $Y := Y' \cup Y''$ ;

**end;**

  output  $(T, Y)$ ;

**end procedure**

At this point, we may as well assume that  $v$  is chosen at random. A specific choice of  $v$  is necessary when  $G$  is a biconnected component of a larger graph (see Section 3.3).

**Lemma 4.** Let  $G$  be biconnected and outerplanar, and let  $v$  denote a vertex in  $G$ . Let  $(T, Y)$  denote the result of the call to **bc-op-td**( $G, v$ ). Then  $(T, Y)$  is a simple tree decomposition of  $G$ , and  $T$  is a spanning tree of  $G - \{v\}$ .

**Proof.** We prove, using induction on  $|E(G)|$ , a somewhat stronger result. We show that  $(T, Y)$  is simple, that  $T$  spans  $G - \{v\}$ , and that for each edge  $ij$  in  $G$ , either  $i \in V(T)$  with  $\{i, j\} \subseteq Y_i$  or  $j \in V(T)$  with  $\{i, j\} \subseteq Y_j$ . The lemma holds for the basis case, in which  $G$  contains just one edge. If  $|E(G)| > 1$ , let  $P$ , with endpoints  $w$  and  $x$ , denote a path that satisfies Lemma 3. Let  $G'$  denote  $G - (P - \{w, x\})$ . Thus  $v$  is in  $G'$ . By the induction hypothesis, **bc-op-td**( $G', v$ ) returns a simple tree decomposition  $(T', Y')$ , with  $T'$  spanning  $G' - \{v\}$ , and with  $\{w, x\} \subseteq Y'_w$  or  $\{w, x\} \subseteq Y'_x$ . Assume, without loss of generality, that  $\{w, x\} \subseteq Y'_w$ . Let  $(T'', Y'')$  denote the  $w$ -extensible tree decomposition of  $P$ . Then  $T'' = P - \{w, x\}$  and  $\{w, x\} \subseteq Y''_a$ , where  $a$  is  $w$ 's neighbor in  $P$ .  $T$  is formed by adding an edge between vertex  $w$  in  $T'$  and vertex  $a$  in  $T''$ . The only vertices common to  $G'$  and  $P$  are  $w$  and  $x$ , which are contained in both  $Y'_w$  and  $Y''_a$ . Therefore  $(T, Y)$  is a valid tree decomposition

<sup>4</sup> The recursive call in this algorithm uses graph difference, whereby  $G_1 - G_2$  denotes the subgraph of  $G_1$  induced by  $V(G_1) - V(G_2)$ .

of  $G$ .  $(T, Y)$  is simple because  $(T', Y')$  and  $(T'', Y'')$  are simple, with the edge  $wa$  existing in  $G$ .  $T'$  spans  $G - \{v\}$  because  $T'$  spans  $G' - \{v\}$  and  $T''$  spans  $P - \{w, x\}$ . To complete the induction, observe that for each edge  $ij \in E(G)$ ,  $\{i, j\} \subseteq Y_i$  or  $\{i, j\} \subseteq Y_j$ , because either  $\{i, j\} = \{w, a\} \subseteq Y_a''$  or  $\{i, j\}$  is contained in one of  $Y_i', Y_j', Y_i''$  and  $Y_j''$ .  $\square$

### 3.2. Time complexity

Let  $G$ , of order  $n$ , and  $v$  denote the input to **bc-op-td**. We store  $G$  in doubly-linked adjacency list format. This is space-efficient, because  $G$  can have at most  $2n - 3$  edges. We also employ a few additional links. To facilitate the removal of an edge  $ab$ , links are maintained between the copy of  $b$  in  $a$ 's adjacency list and the copy of  $a$  in  $b$ 's adjacency list. The only steps in **bc-op-td** that take more than constant time are (i) finding a path  $P$  that satisfies Lemma 3, (ii) deleting the edges and non-endpoint vertices of  $P$  from  $G$  and (iii) constructing an extensible tree decomposition of  $P$ . Of these, steps (ii) and (iii) take at most linear time over all calls to **bc-op-td**. Thus the question of efficiency reduces to the implementation of step (i). One fast method is described below.

Some preprocessing is required. We first construct an outerplanar layout of  $G$  and find the Hamiltonian circuit that constitutes its external face [4]. We scan the layout in a clockwise direction, starting at  $v$ , and number vertices in the order in which they are encountered. Then we rearrange the adjacency list of each vertex so that the elements in the list are numbered in ascending order. (An efficient means to do this visits the vertices in order; for each vertex  $a$  and each vertex  $b$  in  $a$ 's adjacency list, we insert  $a$  into  $b$ 's adjacency list in a new structure.) Each of these tasks takes only linear time.

Once preprocessing is completed, paths to play the role of  $P$  are found during a second clockwise scan and calls to procedure **find-path**. For  $1 \leq i \leq n$ , let  $v_i$  denote the vertex numbered  $i$ . A variable  $k$ , initialized to 1, stores the number of the last vertex scanned. During each call to **find-path**, the scan starts from this vertex and continues until a satisfactory path is found. As paths are found, non-endpoint vertices are removed (by **bc-op-td**), but the original vertex numbering is maintained. A stack  $vstack$ , initially empty, holds previously scanned vertices of degree three or more

(these vertices are candidates for the endpoints of paths). We say that two vertices are *internal neighbors* if they are connected by an internal edge.

#### Procedure find-path

**Input:** A biconnected outerplanar graph  $G$  of order three or more with  $V(G) \subseteq \{v_1, \dots, v_n\}$ , a stack  $vstack$  of vertices, and an integer  $k$ .

**Output:** A path  $P$  that satisfies Lemma 3, with  $v_1$  playing the role of  $v$ .

#### begin procedure

if  $G$  is a cycle

then  $P := G - \{v_1 v_n\}$ ;

else begin

while  $v_k$  has no lower-numbered internal neighbor

do begin

if  $\delta(v_k) \geq 3$  then push  $v_k$  on to  $vstack$ ;

$k := k + 1$ ;

end;

$v_j :=$  the vertex on top of  $vstack$ ;

$P :=$  the path on the external face from  $v_j$  to  $v_k$ ;

if  $\delta(v_j) = 3$  then pop  $v_j$  from  $vstack$ ;

end;

output  $P$ ;

end procedure

**Lemma 5.** A path returned by **find-path** satisfies Lemma 3.

**Proof.** Let  $G$ ,  $P$ ,  $j$  and  $k$  be as defined in **find-path**. The lemma holds immediately if  $G$  is a cycle, so assume  $G$  contains one or more internal edges. Since  $1 \leq j < k$ ,  $P$  does not contain  $v_1$  as a non-endpoint vertex. Let  $v_a$  denote the lowest-numbered internal neighbor of  $v_k$ . Clearly  $a < k$ . Let  $v_b$  denote the highest-numbered internal neighbor of  $v_j$ . Since  $v_j$  was on the stack, it had no lower-numbered internal neighbors when  $v_k$  was scanned, and so  $j < b$ . For  $v_i$ , a non-endpoint vertex of  $P$ ,  $\delta(v_i) = 2$ , else  $v_i$  would either have been pushed on  $vstack$  after  $v_j$ , or a path connecting  $v_j$  and  $v_i$  would already have been returned. Thus  $a \leq j$  and  $k \leq b$ . It cannot be that both  $a < j$  and  $k < b$ , because otherwise the edges  $v_a v_k$  and  $v_j v_b$  would intersect, which is impossible in an outerplanar layout. So either  $a = j$  or  $k = b$ , and the internal edge  $v_j v_k$  must exist. Thus  $P$  satisfies all three properties required by Lemma 3: its non-

endpoint vertices all have degree two, its endpoints are adjacent, and it excludes  $v_1$  (which plays the role of  $v$ ) as a non-endpoint vertex.  $\square$

The total amount of work done by **find-path** is linear in  $n$ . To see this, observe that the number of iterations of the while loop, summed over all calls to **find-path**, is at most  $n$ . Thus each statement in **find-path** is executed  $O(n)$  times. The only statement that takes more than constant time is the assignment of a value to  $P$ , which takes  $O(|V(P)|)$  time. Since the order of  $G$  decreases by  $O(|V(P)|)$  after each such assignment, the total amount of work done during this operation is also  $O(n)$ . It follows that **bc-op-td** has linear complexity.

### 3.3. Tackling general outerplanar graphs

We now generalize our algorithm to handle all outerplanar graphs.

#### Procedure op-td

**Input:** An outerplanar graph  $G$  of order two or more, and sets  $B$  and  $C$  of its biconnected components and cut points.

**Output:** A simple tree decomposition  $(T, Y)$  of  $G$ , with  $T$  spanning  $G$ .

**begin procedure**

  if  $G$  is biconnected

**then begin**

$u, v :=$  any two adjacent vertices in  $G$ ;

$(T', Y') := \text{bc-op-td}(G, v)$ ;

$Y_v = \{v\}, T := T' \cup \{v\} \cup \{uv\}, Y := Y' \cup \{Y_v\}$ ;

**end;**

**else begin**

$B_i :=$  an element of  $B$  that contains exactly one vertex  $v$  from  $C$ ;

**if**  $v$  is not a cut point in  $G - (B_i - \{v\})$  **then**

$C := C - \{v\}$ ;

$(T', Y') := \text{bc-op-td}(B_i, v)$ ;

$(T'', Y'') :=$

$\text{op-td}(G - (B_i - \{v\}), B - \{B_i\}, C)$ ;

$u :=$  an arbitrary neighbor of  $v$  in  $B_i$ ;

$T := T' \cup T'' \cup \{uv\}, Y := Y' \cup Y''$ ;

**end;**

      output  $(T, Y)$ ;

**end procedure**

**Lemma 6.** *Let  $G$  be outerplanar. Let  $(T, Y)$  denote the result of the call to **op-td**( $G$ ). Then  $(T, Y)$  is a simple spanning tree decomposition of  $G$ .*

**Proof.** The proof proceeds by induction on the number of biconnected components of  $G$ . The basis case, when  $G$  is biconnected, follows from Lemma 4 and the modifications made to  $(T, Y)$  after the call to **bc-op-td**( $G, v$ ). So let  $B_i, v, (T', Y'), (T'', Y'')$  and  $u$  be as defined in **rop-td**. Let  $\hat{G}$  denote  $G - (B_i - \{v\})$ . From the proof of Lemma 4, we know that  $(T', Y')$  is simple, that it spans  $B_i - \{v\}$ , and that  $\{u, v\} \subseteq Y'_u$  (there is no  $Y'_v$ ). By the induction hypothesis,  $(T'', Y'')$  is a simple spanning tree decomposition of  $\hat{G}$ . Thus, by construction,  $(T, Y)$  is a simple spanning tree decomposition of  $G$ .  $\square$

Biconnected components and cut points can be found using a depth-first search. Procedure **op-td** builds an optimal tree-decomposition using **bc-op-td**. This tree decomposition is converted into a path decomposition using **td2pd**. Recalling Theorem 1, and noting that **td2pd** is asymptotically the slowest of the aforementioned steps, we achieve the following main result.

**Theorem 7.** *If  $G$  is an outerplanar graph of order  $n$ , a path decomposition of  $G$  with width at most  $3 \times pw(G) + 2$  can be constructed in  $O(n \log n)$  time.*

We strongly suspect that this bound can be reduced to  $O(n)$ . After all, we have gone to some effort here to devise linear-time techniques. Only computing the path decomposition of a tree uses super-linear time. We are aware that claims of linear-time path decomposition algorithms for trees have appeared in the literature; but none to our knowledge has a credible level of detail, such as that found in [5].

## 4. Concluding remarks

We have implemented our algorithm in the C programming language. Tests on a SPARC ULTRA indicate that the implementation is fast in practice, taking, for instance, less than two seconds to compute the path decomposition of a graph with ten thousand vertices. It is difficult to gauge the quality of the solutions produced, because there is no practical way

to obtain optimal path decompositions for comparison. As a compromise, we tested the program on pseudo-random outerplanar graphs of known pathwidth. These tests indicate that the approximate decompositions tend to have much smaller width than the worst case guarantee.

Our work has exploited the fact that if the width of a tree decomposition  $(T, Y)$  of  $G$  is bounded, and if  $pw(T)$  is within some constant multiple of  $pw(G)$ , then we can construct a path decomposition of  $G$  whose width is at most a constant times  $pw(G)$ . Series-parallel graphs also have treewidth at most two. Optimal tree decompositions for them can be constructed quickly. We believe that, for these graphs, it is possible to ensure  $pw(T) \leq 2pw(G)$ , yielding a factor-of-six relative approximation algorithm.

On a more general note, we conjecture that any graph  $G$  has an optimal tree decomposition  $(T, Y)$  such that  $pw(T) \leq pw(G)$ . If true, a constructive proof of this would provide a relative approximation algorithm for any class of graphs whose bounded-width tree decompositions can be found efficiently. Currently, this class includes all graphs of treewidth four or less, Halin graphs and, for any fixed  $k$ ,  $k$ -chordal graphs,  $k$ -outerplanar graphs and graphs with disk dimension  $k$ , to name just a few [2,6,7].

## References

- [1] H.L. Bodlaender, T. Kloks, Efficient and constructive algorithms for the pathwidth and treewidth of graphs, *J. Algorithms* 25 (1996) 358–402.
- [2] H.L. Bodlaender, Classes of graphs with bounded treewidth, Technical Report RUU-CS-86-22, University of Utrecht, The Netherlands, December 1986.
- [3] N. Deo, M.S. Krishnamoorthy, M.A. Langston, Exact and approximate solutions for the gate matrix layout problem, *IEEE Trans. Computer-Aided Design* 6 (1987) 79–84.
- [4] J. Ellis, M. Mata, G. MacGillivray, Linear time algorithms for longest  $(s, t)$ -paths and longest circuits in weighted outerplanar graphs, Technical Report DCS-65-IR, Department of Computer Science, University of Victoria, Victoria, BC, August 1987.
- [5] J.A. Ellis, I.H. Sudborough, J.S. Turner, The vertex separation and search number of a graph, *Inform. and Comput.* 113 (August 1994) 50–79.
- [6] M.R. Fellows, M.A. Langston, On well-partial-order theory and its application to combinatorial problems of VLSI design, *SIAM J. Discrete Math.* 5 (1) (1992) 117–126.
- [7] R. Govindan, Algorithmic methods for circuit layout and partitioning, Ph.D. Thesis, Department of Computer Science, University of Tennessee, Knoxville, TN, 1998.
- [8] T. Kashiwabara, T. Fujisawa, NP-completeness of the problem of finding a minimum-clique-number interval graph containing a given graph as a subgraph, in: *Proc. International Symposium on Circuits and Systems*, 1979, pp. 657–660.
- [9] N.G. Kinnersley, The vertex separation of a graph equals its path-width, *Inform. Process. Lett.* 42 (1992) 345–350.
- [10] A. Kornai, Z. Tuza, Narrowness, pathwidth, and their application in natural language processing, *Discrete Appl. Math.* 36 (1992) 87–92.
- [11] N. Robertson, P.D. Seymour, Graph minors II. Algorithmic aspects of treewidth, *J. Algorithms* 7 (1986) 309–322.
- [12] X. Yan, A relative approximation algorithm for computing pathwidth, Master's Thesis, Department of Computer Science, Washington State University, Pullman, WA, 1989.



**ContentsDirect**, the *free* e-mail alerting service, delivers the table of contents for this journal directly to your PC, prior to publication. The quickest way to register for **ContentsDirect** is via the Internet at: <http://www.elsevier.nl/locate/ContentsDirect>. If you don't have access to the Internet you can register for this service by sending an e-mail message to [cdsubs@elsevier.co.uk](mailto:cdsubs@elsevier.co.uk) – specifying the title of the publication you wish to register for.

### Editorial Policy and Scope

The aim of *IPL* is to allow rapid dissemination of interesting results in the field of information processing in the form of short, concise papers. To this end, submissions should not exceed the equivalent of nine A4 or 8 1/2-by-11 double-spaced typed pages.

The scope of the journal is indicated by the list of keywords found below. This list is periodically updated by inserting items most frequently proposed by the contributors and removing the least popular entries, under the advisement of the Board of Editors. Submissions are encouraged both on theoretical work and on experimental work.

The scope of *IPL* is suggested by the following alphabetical list of keywords:

Algorithms – analysis of algorithms – automatic theorem proving – combinatorial problems – compilers – computational complexity – computational geometry – computer architecture – concurrency – cryptography – databases – data structures – design of algorithms – distributed computing – distributed systems – fault tolerance – formal languages – formal semantics – functional programming – information retrieval – interconnection networks – operating systems – parallel algorithms – parallel processing – performance evaluation – program correctness – program derivation – programming calculi – programming languages – program specification – real-time systems – safety/security in digital systems – software design and implementation – software engineering – specification languages – theory of computation.

### Instructions for Authors

Contributions should be sent *in quadruplicate* to a member of the Board of Editors selected for reasons of geographic or subject-area proximity. Contributions must be in English or American. Since no linguistic assistance in the form of copy editing is provided by *IPL*, poorly written papers will not be considered.

The length of a contribution (including all figures, references etc.) should not exceed the equivalent of nine A4 or 8 1/2-by-11 pages, typed with wide margins and double line-spacing, i.e. approximately a 20K character file or 3,000 words. An Editor's decision to reject a contribution because of its length is final.

An abstract is not required, but *appropriate keywords must be supplied*; these should include at least one keyword drawn from the list of keywords given above. References should be numbered and put in *alphabetical* order at the end of the paper. References should contain the names and initials of all authors, title of the article, name of the journal, volume number, year of publication (or title of volume, name of editor(s), name of publishers), and page numbers. See papers in existing issues of *IPL* for the preferred style.

Since *IPL* is not published from camera-ready copies, there is no need to supply fancy print-outs. If computer-assisted printing is used, care must be taken that the print is legible and properly paginated and that the paper satisfies the length constraint stated earlier. Figures should be sharp *glossy prints* of about manuscript size or, preferably, *original drawings* or computer output.

Once a submission has been accepted for publication, *all further correspondence should be sent directly to the Publisher* (Elsevier Science B.V., P.O. Box 2759, 1000 CT Amsterdam, The Netherlands – refer to *IPL* on the envelope). Since all proofreading is done by the Publisher's staff and *proofs are not sent to the author*, the presentation of the manuscript should be extremely clear. Handwritten Greek letters and unusual symbols should be identified in the margin the first time they occur in the text.

Upon acceptance of an article, author(s) are asked to transfer copyright of the article to the Publisher. This transfer of copyright will ensure the widest possible dissemination of information.

There are no page charges. Fifty free offprints are supplied for each article published. Further offprints can be ordered from the Publisher. (Both an Offprint Order Form and a Copyright Transfer Notice will be sent by the Publisher upon receipt of the paper.)

### Instructions for LaTeX manuscripts

The LaTeX files of papers that have been accepted for publication may be sent to the Publisher by email or on a diskette (3.5" or 5.25" MS-DOS). If the file is suitable, proofs will be produced without rekeying the text. The article should be encoded in Elsevier-LaTeX, standard LaTeX, or AMS-LaTeX (in document style "article"). The Elsevier-LaTeX package (including detailed instructions for LaTeX preparation) can be obtained from the Comprehensive TeX Archive Network (CTAN). Search for Elsevier on the CTAN Search page (<http://www.ucc.ie/cgi-bin/ctan/>), or the CTAN-Web page (<http://tug2.cs.umb.edu/ctan/>), or use direct access via FTP at <ftp.dante.de> (Germany), <ftp.tex.ac.uk> (UK), or <tug2.cs.umb.edu> (Massachusetts, USA) and go to the directory `/tex-archive/macros/latex/contrib/supported/elsevier`. No changes from the accepted version are permissible, without the explicit approval by the Editor. The Publisher reserves the right to decide whether to use the author's file or not. If the file is sent by email, the name of the journal, *IPL*, should be mentioned in the "subject field" of the message to identify the paper. Authors should include an ASCII table (available from the Publisher) in their files to enable the detection of transmission errors.

The files should be mailed to: Ms. Paulette de Boer, Elsevier Science B.V., P.O. Box 2759, 1000 CT Amsterdam, The Netherlands, Email: [p.boer@elsevier.nl](mailto:p.boer@elsevier.nl).

### Author's Benefits

- (1) 50 offprints per contribution free of charge.
- (2) 30% discount on all Elsevier Science books.

---

### Publishing Staff

#### Issue Manager

Elma Kleikamp  
Tel.: +31 20 485 2640  
[e.kleikamp@elsevier.nl](mailto:e.kleikamp@elsevier.nl)

### Product Manager

Arjen Sevenster

Elsevier Science B.V.  
P.O. Box 2759  
1000 CT Amsterdam  
The Netherlands

# Stable set and multiset operations in optimal time and space \*

Bing-Chao Huang

ESP Group, Incorporated, 40063 Fremont Boulevard, Fremont, CA 94538-6045, USA

Michael A. Langston

Department of Computer Science, University of Tennessee, Knoxville, TN 37996-1301, USA

Communicated by F. Dehne

Received 28 March 1991

Revised 15 May 1991

## Abstract

Huang, B.-C. and M.A. Langston, Stable set and multiset operations in optimal time and space, Information Processing Letters 39 (1991) 131–136.

We devise time-space optimal methods for stably performing set and multiset operations on sorted files of data. For the sake of complete generality, our techniques neither modify records nor require any information other than a record's key.

**Keywords:** Analysis of algorithms, computational complexity, data management

## 1. Introduction

Rearranging the sequence of records within a file based on the relative value of each record's key is an operation of fundamental importance to computer science. Common examples of this type of operation include sorting a random file, extracting duplicates from a sorted file, merging two or more sorted subfiles, and many others. It is often desirable that such an operation be *stable*, by which we mean that records with equal keys retain their original relative order.

Performing such an operation in the optimal amount of time (to within a constant factor) is usually not very difficult, but the obvious methods incur a considerable cost in the overhead associated with additional temporary storage to be used by the fast rearrangement algorithm. On the other hand, slower methods typically exist for performing such an operation in-place, where a few extra storage cells aid the rearrangement process, but whose total number is constant, independent of the file's size.

The design and analysis of *time-space optimal* file rearrangement algorithms, those that achieve lower bounds on both time and extra space simultaneously, is an appealing area of research, so far largely unexplored. Through the work of [8], it is known that stable merging (and, hence, stable sorting by merging) permits such a method. More recently, the authors have shown that stable duplicate-key extraction does as well [2].

\* A preliminary version of this paper was presented at the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems held in Austin, Texas, in March, 1988. This research has been supported in part by the National Science Foundation under grant MIP-8919312 and by the Office of Naval Research under contract N00014-88-K-0343.

The goal of this brief paper is to prove the existence of time-space optimal methods for the elementary binary set and multiset operations<sup>1</sup> on sorted files. In the next section, we introduce some required notation and define a few useful, primitive suboperations. Section 3 contains our presentation of a time-space optimal selection scheme, in which matched records are selected from a sorted file. Primarily through the use of this selection strategy, we prove in Section 4 that each set operation can be performed on sorted files of data in optimal time and space. In Section 5, we generalize these operations to multisets, with special attention to file processing applications, and show that each can again be performed in optimal time and space. The final section of this paper contains a few remarks pertinent to the future study of this general topic, and provides upper bounds on the constants of proportionality of our methods.

## 2. Notation, definitions and other relevant preliminaries

Let  $L$  denote a list (internal file) of  $n$  records, indexed from 1 to  $n$ . We use  $KEY(i)$  as a shorthand to denote the key of the record with index  $i$ . Only the two common  $O(1)$  time and space primitive operations are assumed, namely, record exchanges and key comparisons. The exchange procedure,  $SWAP(i, j)$ , directs that the  $i$ th and  $j$ th records are to be exchanged. The comparison functions, for example  $KEY(i) < KEY(j)$ , return the expected Boolean values dependent on the relative values of the keys being compared.

From these primitive operations, we construct a few  $O(1)$  space useful subprograms for dealing with *blocks*. Let us define a block to be a set of records from  $L$  with consecutive indices. The *head* of a block is the record with the lowest index (or, informally, the "leftmost" record in the block). The procedure  $BLOCKSWAP(i, j, h)$  exchanges a block of  $h$  records beginning at index  $i$  with a

block of  $h$  records beginning at index  $j$  in  $O(h)$  time. We specify that blocks do not partially overlap (i.e., if  $i \neq j$ , then  $h \leq |i - j|$ ) and that, when  $BLOCKSWAP$  is finished, records within a moved block retain the order they possessed before  $BLOCKSWAP$  was invoked. A block of  $h$  records beginning at index  $i$  is sorted in nondecreasing order by the procedure  $SORT(i, h)$ . The procedure  $BLOCKSORT(i, h, p)$  uses  $BLOCKSWAP$  to rearrange the  $p$  consecutive blocks, each with  $h$  records, beginning at index  $i$  so that their heads are sorted in nondecreasing order. To reduce unnecessary record movement, we assume  $SORT$  and  $BLOCKSORT$  use a straight selection sort [4], yielding respective time complexities  $O(h^2)$  and  $O(p^2 + ph)$ . Finally, the procedure  $ROTATE(i, h, l)$  rotates (circularly shifts) a block of  $h$  records, beginning at index  $i$ ,  $l$  places to the left. We assume  $ROTATE$  is implemented in the common fashion with three sublist reversals, thereby requiring no more than  $h$  invocations of  $SWAP$ .

## 3. Selection: A fundamental file processing tool

Suppose  $L$  contains two sorted sublists,  $L_1$  and  $L_2$ , with respective sizes  $n_1$  and  $n_2$ , where  $n_1 + n_2 = n$ . We seek to transform stably  $L_1$  into two sorted sublists  $L_3$  and  $L_4$ , where  $L_3$  contains the records whose keys are not found in  $L_2$ , and  $L_4$  contains those whose keys are. Thus we accept  $L = L_1L_2$ , and  $SELECT$  keys from  $L_1$  that are contained in  $L_2$ , and accumulate them in  $L_4$ , where our output is of the form  $L_3L_4L_2$ .

Any procedure for performing  $SELECT$  requires  $\Omega(n)$  time since this problem is at least as difficult as the task of verifying whether  $L_1$  and  $L_2$  contain the same keys, which needs  $\Omega(n)$  comparisons. Similarly, any procedure for  $SELECT$  trivially requires  $\Omega(1)$  extra space. Unfortunately, the obvious methods for attaining either lower bound alone violate the other bound.

However, we shall now proceed to prove that both lower bounds can be simultaneously achieved. In order to do so, we shall employ the concepts of *internal buffering* and *block rearranging*. This general type of approach, in which  $O(\sqrt{n})$  blocks each

<sup>1</sup> These operations are commonly defined to be *union*, *intersection*, *difference* (relative complement) and *exclusive OR* (symmetric difference).

of size  $O(\sqrt{n})$  are exchanged, can be traced back to the seminal work on unstable merging described in [5].

We note that, as with the other results proved here, it is sufficient to consider only sorted inputs. That is, it is easy to show that any algorithm for performing *SELECT* when  $L_1$  and  $L_2$  are unsorted needs  $\Omega(n \log n)$  time, because otherwise a faster algorithm could be employed to solve set equality, contradicting the main result of [7]. Thus dealing with unsorted inputs is not an issue, since we know from [8] that we can stably sort in  $O(n \log n)$  time and  $O(1)$  extra space.

**Theorem 1.** *SELECT can be stably performed simultaneously in linear time and constant extra space.*

**Proof.** Our proof is by means of algorithm construction, emphasizing simplicity of exposition rather than efficiency of implementation. We shall henceforth use the term  $L_3$  record ( $L_4$  record) to denote a record initially in  $L_1$  that is to go to  $L_3$  ( $L_4$ ). Similarly, the term  $L_3$  block ( $L_4$  block) shall be used to refer to a block containing only  $L_3$  ( $L_4$ ) records.

*Step 1. (BUFFERFILL)*

Let  $s = \lfloor \sqrt{n_1} \rfloor$ . We first attempt to fill an internal buffer of size  $s$  with records having the (first copy of the)  $s$  smallest distinct keys that are to go to  $L_4$ . Thus we seek to convert  $L_1$  into the form  $ABC$ , where  $B$  is the buffer (whose contents need be in no special order),  $A$  results from obtaining  $B$ , and  $C$  is a suffix of  $L_1$  whose records have not been disturbed. We construct  $B$  by conducting a left-to-right scan of  $L_1$  in conjunction with a left-to-right scan of  $L_2$ . When a comparison of adjacent keys in  $L_1$  reveals that a record  $R$  with a new distinct key has been found, and when  $L_2$  indicates that  $R$  is also to go to  $L_4$ , then we coalesce  $R$  into  $B$ . That is, we increase the size of  $B$  by 1, and advance the scan to the record to the immediate right of  $R$ . Otherwise, we exchange  $R$  with the current leftmost element of  $B$ , thereby shifting  $B$  one position to the right and leaving its size unchanged. Therefore, we "roll"  $B$  toward the

right in an attempt to load it with  $s$  records having distinct keys.

If  $L_1$  is exhausted before  $B$  is filled, then we go immediately to Step 4. Otherwise, as soon as  $B$  has grown to size  $s$ , we *ROTATE* the sublist  $BC$  to yield the list  $ACBL_2$ .

Since only  $B$  has become disordered, and since  $B$  contains only distinct keys, sorting  $B$  in a subsequent step will ensure stability. Clearly,  $O(n)$  time and  $O(1)$  space suffice for the first step.

*Step 2. (BLOCKIFY)*

Our next step is the most complex. Let  $n_3$  ( $n_4$ ) denote the number of records in  $L_1$  that are to go to  $L_3$  ( $L_4$ ), where  $n_1 = n_3 + n_4$ . (We assume that  $n_3$  and  $n_4$  were precomputed with a linear scan of  $L_1$  and  $L_2$  and that, without loss of generality, both quantities are nonzero.) Thus  $n_3 = st_3 + e_3$  and  $n_4 = st_4 + e_4$  for unique nonnegative integers  $t_3, e_3, t_4$ , and  $e_4$ , where  $e_3, e_4 < s$ . We now seek to use  $B$  to transform  $ACB$  into a list of  $t_3 + t_4 + 2$  blocks, the first a (possibly empty) block of size  $e_3$ , followed by  $t_3 + t_4$  blocks of size  $s$ , followed by one last (possibly empty) block of size  $e_4$ , where the first  $t_3 + 1$  blocks contain  $L_3$  and the next  $t_4 + 1$  blocks contain  $L_4$ .

Viewing  $ACB$  in this form, we let the rightmost nonempty block (which has size  $e_4$  or, if  $e_4 = 0$ , size  $s$  and which is now occupied by  $B$  or a part of  $B$ ) become an  $L_4$  block. The block to its immediate left will become an  $L_3$  block. (Recall that both  $n_3$  and  $n_4$  exceed zero, so that these two blocks are needed.) We now begin scanning  $AC$  and  $L_2$  from right to left. When we find an  $L_4$  record, we exchange it with the rightmost buffer element of the current  $L_4$  block. For an  $L_3$  record, we exchange it with the rightmost buffer element of the current  $L_3$  block (unless, of course, the current  $L_3$  block contains no buffer elements). Therefore, the buffer is, in general, broken into two pieces, each to the left of the growing edge of a block. Should an  $L_3$  block be filled before the current  $L_4$  block is completed, then we simply begin a new  $L_3$  block to the immediate left of the newly-filled  $L_3$  block. When the current  $L_4$  block is filled, in which case the buffer now occupies  $s$  contiguous record locations, we begin a new  $L_4$  block to the immediate left of the current  $L_3$  block. At this point, we reverse the new-block

rule. (That is, we begin a new  $L_4$  block to the immediate left of a newly filled  $L_4$  block until the current  $L_3$  block is filled, at which time the buffer again occupies  $s$  contiguous record locations and we begin a new  $L_3$  block to the immediate left of the current  $L_4$  block.) We continue this process of building blocks and reversing the new-block rule as necessary until all of  $AC$  has been examined.

If  $e_3 = 0$ , then the buffer now occupies the leftmost nonempty block, which has size  $s$ . If  $e_3 > 0$ , then we use *BLOCKSWAP* to exchange the  $e_3$  buffer records in the leftmost nonempty block with the rightmost  $e_3$  records in the current  $L_3$  block, thereby consolidating the buffer into one  $s$ -sized block. At this time, the leftmost block of  $L_3$  (of size  $e_3$ ) is finished.

Again,  $B$  is the only block that is internally disordered, insuring stability.  $O(n)$  time and  $O(1)$  space are sufficient for this step as well.

#### Step 3. (BLOCK REARRANGEMENT)

$L_3$  is now made up of a sequence of blocks interspersed with another sequence constituting  $L_4$ . Except for the buffer, each sequence of blocks as well as the elements within each block are in the proper order. We now use the buffer to separate the two sequences.

We invoke *SORT* on the buffer and, since our choice of  $s$  ensures that  $t_3 + t_3 \leq s$ , use the buffer to "remember" the two interspersed sequences. That is, we use a scan of  $L_2$  to classify each block as type  $L_3$  or  $L_4$ , exchanging the  $t_3$  smallest buffer elements with the respective heads of the  $t_3$   $s$ -sized  $L_3$  blocks, and exchanging the  $t_4 - 1$  largest buffer elements with the appropriate heads of the  $t_4 - 1$   $s$ -sized  $L_4$  blocks other than  $B$ . We next use *BLOCKSWAP* to move the buffer to the position of the rightmost  $s$ -sized block. Then we invoke *BLOCKSORT* on the first  $t_3 + t_4 - 1$   $s$ -sized blocks and restore the contents of the buffer. Finally, if  $e_4 > 0$ , we use *BLOCKSWAP* to exchange the  $e_4$   $L_4$  records to the immediate right of the buffer with the leftmost  $e_4$  records in the buffer. Thus we have produced  $L_3(L_4 - B)BL_2$ , and now go to Step 5.

Since both *SORT* and *BLOCKSORT* operate only on distinct keys, our actions at this step are stable. Also,  $O(n)$  time and  $O(1)$  space suffice, since each sort involves  $O(\sqrt{n})$  keys.

#### Step 4. (SPECIAL CASE)

Suppose there are only  $s' < s$  distinct-keyed records to go to  $L_4$ . Thus Step 1 has replaced  $L_1$  with  $AB$ , where  $B$  is smaller than desired. (Without loss of generality, we assume that  $s'$  is nonzero.) Since bringing the buffer's size up to  $s$  with duplicate keys would eliminate stability, we adopt the following strategy that uses larger blocks of size  $b = \lceil n_1/s' \rceil$ . Thus,  $n_1 - s' = bt = e$  for unique nonnegative integers  $t$  and  $e$ , where  $e < b$ . We now seek to transform  $A$  into a list of  $t$  blocks, the first a block of size  $b + e$ , followed by  $t - 1$  blocks of size  $b$ .

Viewing  $A$  in this form, we compare its rightmost two blocks. If the number of  $L_4$  records is as large as the number of  $L_3$  records in these two blocks, then we *ROTATE* each consecutive segment of  $L_4$  records with  $L_3$  records to their right as necessary so that the rightmost block is of type  $L_4$ . If there are more  $L_3$  records present, we do likewise to make the rightmost block of type  $L_3$ . Having finished the rightmost block, we next consider the pair of blocks to its immediate left, and so on. We repeat this process on adjacent pairs of blocks until there is only one block left, of size  $b + e$ , whose elements we rotate as necessary to form an  $L_3$  block  $D$  followed by an  $L_4$  block  $E$ . Thus  $A$  has taken on the form  $DEF$ , where  $F$  contains  $t - 1$   $b$ -sized blocks, each of type  $L_3$  or  $L_4$ . Now, in a fashion analogous to that of the general case described in Step 3, we *SORT* the buffer and use it to "remember" the two sequences interspersed in  $F$  (note that this is possible since our choice of  $b$  ensures that  $s' > t$ ). Then we invoke *BLOCKSORT* to transform  $F$  into  $GH$  where  $G$  ( $H$ ) contains only  $L_3$  ( $L_4$ ) blocks, and restore the contents of the buffer. Finally, we *ROTATE* the sublist  $EG$  to obtain  $L_3(L_4 - B)BL_2$ .

As with Steps 2 and 3 of the general algorithm, our actions at Step 4 are stable. Since there are only  $s'$  distinct keys to go to  $L_4$ , blockifying the list needs only  $O(s')$  rotations, each of length  $O(b)$ , guaranteeing the  $O(n)$  time and  $O(1)$  space bounds. As in Step 3 of the general algorithm, sorting blocks and sorting the buffer involve at most  $O(\sqrt{n})$  keys and need only  $O(n)$  time and  $O(1)$  space.

#### Step 5. (FINISH UP)

From the left-to-right scan that produced  $B$  in Step 1, we know  $B$  contains the leftmost record from  $L_1$  containing each distinct key that is represented in  $B$ , so that we can achieve stability. Therefore, we now simply scan both  $L_4 - B$  and  $B$  from right to left, merging then by employing *ROTATE* on the rightmost unmerged sequences of  $L_4 - B$  and the appropriate leftmost unmerged sequences of  $B$  to produce  $L_3 L_4 L_2$ , the desired result.  $O(n)$  time and  $O(1)$  space suffice for this final step as well, since records from  $L_4 - B$  are rotated at most once while those from  $B$ , of which there are only  $O(\sqrt{n})$ , are moved at most  $O(\sqrt{n})$  times.  $\square$

#### 4. Set operations

In what follows, suppose we are given the input list  $L = XY$ , where  $X$  and  $Y$  are two sublists, each sorted on the key, and each containing no duplicates. Since the same key may naturally appear once in  $X$  and once in  $Y$ , we insist that, in the spirit of stability, the record represented in the result of a binary set operation be the one that occurs first in  $L$ . As with the problem of selection addressed in the last section, the elementary binary set operations union, intersection, difference and exclusive *OR* are each at least as difficult as verification, and therefore require  $\Omega(n)$  time and, of course,  $\Omega(1)$  space.

**Theorem 2.** *Set union, intersection, difference and exclusive OR can be stably performed simultaneously in both linear time and constant extra space.*

**Proof.** The tools now at hand make the proof easy. We first identify these fundamental tools (each of which is stable and requires only  $O(n)$  time and  $O(1)$  extra space). From [8], we obtain *MERGE*. From [2], we obtain *DUPLICATE-KEY EXTRACT*. From the work in the previous section, we obtain *SELECT*.

We invoke *MERGE* followed by *DUPLICATE-KEY EXTRACT* to produce  $X \cup Y$ . We perform *SELECT* to yield both  $X \cap Y$  and  $X - Y$ . To achieve  $X \oplus Y$ , we invoke *SELECT* on  $XY$

producing  $X_1 X_2 Y$ , *ROTATE*  $X_2$  and  $Y$  yielding  $X_1 Y X_2$ , perform *SELECT* on  $Y X_2$  producing  $X_1 Y_1 Y_2 X_2$ , and finally *MERGE*  $X_1$  and  $Y_1$ .  $\square$

#### 5. Multiset operations

For file processing applications, a list or sublist may, of course, contain a number of records with the same key. How then are multiset operations to be defined? Curiously, there is no definitive set theory answer, for example, as to whether  $\{1, 2, 2\} \cap \{2, 2, 3\}$  is  $\{2\}$ ,  $\{2, 2\}$  or  $\{2, 2, 2, 2\}$ . While the issue is seldom even addressed in the literature, at least one source [6] has mentioned the second interpretation. However, we suggest that the third interpretation may be particularly reasonable for file operations, since a record typically contains more information than the key alone.

Suppose we are given the input list  $L = XY$ , where  $X$  and  $Y$  are two sublists, each sorted on the key. We define multiset operations as follows:

*union*  $X \cup Y \equiv$  the stably sorted list containing all records in  $X$  and all records in  $Y$ ,

*intersection*  $X \cap Y \equiv$  the stably sorted list containing all records whose keys are in both  $X$  and  $Y$ ,

*difference*  $X - Y \equiv$  the stably sorted list containing all records whose keys are in  $X$  but not  $Y$ ,

*exclusive OR*  $X \oplus Y \equiv$  the stably sorted list containing all records whose keys are in  $X$  or  $Y$ , but not both.

We observe that, with multiset operations as defined above, De Morgan's law holds. Also, each operation requires  $\Omega(n)$  time and  $\Omega(1)$  extra space.

**Theorem 3.** *Multiset union, intersection, difference and exclusive OR can be stably performed simultaneously in both linear time and constant extra space.*

**Proof.** Again, the tools now at hand make the proof easy. We use *MERGE* to produce  $X \cup Y$ . We invoke *SELECT* to yield  $X - Y$ . For the remaining two operations, we first perform the *SELECT-ROTATE-SELECT* series of primitives described in the proof of Theorem 2 to

produce  $X_1Y_1Y_2X_2$ . We *MERGE*  $X_1$  and  $Y_1$  to obtain  $X \oplus Y$ . To achieve  $X \cap Y$ , we *ROTATE*  $Y_2X_2$  to obtain  $X_2Y_2$ , then *MERGE*  $X_2$  and  $Y_2$ .  $\square$

While our interpretations have been chosen for consistency in file processing applications, we observe that the interpretations of [6] equate union with the maximum number of occurrences, intersection with the minimum number of occurrences, and so on. Thus De Morgan's law holds for these interpretations as well, with the binary set operations reducing to special cases of these multiset definitions. (Stability, however, now seems to have no obvious meaning.) By modifying slightly the way in which our *SELECT* algorithm scans  $L_2$  to determine which records go to  $L_4$ , we note that each multiset operation of [6] can also be performed simultaneously in both linear time and constant extra space.

## 6. Remarks

Given today's low cost of computer memory components, it may be that these methods can be used to best advantage when working with large external files. In such an environment, in-place operations can be viewed as a means for increasing the effective size of internal memory, which can result in reduced *I/O* time (through fewer transfers, more and larger buffers, etc.), and a corresponding drastic reduction in the overall elapsed time for file processing. Furthermore, we think it is likely that each time-space optimal operation can be performed *directly* (as with the proof of Theorem 1) and in *parallel* (as with the merge and sort routines recently devised in [1]).

To gauge the practical potential of these time-space optimal schemes, we briefly focus on key comparisons and record exchanges. These two fundamental operations are usually regarded as by far the most time consuming for internal file processing. Both require storage-to-storage instructions for most architectures. Moreover, it is possible to count them independently from the code of any particular implementation. Therefore,

Table 1  
Constant of proportionality upper bounds

Operation	Set	Multiset
Union	13	7
Intersection	11.5	32
Difference	11.5	11.5
Exclusive OR	31	31

their total gives a useful estimate of the size of the linear-time constant of proportionality for each algorithm presented.

These worst-case totals are shown in Table 1. The computation of each value <sup>2</sup> displayed is straightforward, and is eliminated from our presentation for the sake of brevity. (For each *MERGE* operation, we use the constant from [3], a marked improvement over that of [8].)

As for the issue of constant extra space, none of our methods explicitly requires more than a couple of dozen additional storage cells for use as pointers, counters and the like.

## References

- [1] X. Guan and M.A. Langston, Time-space optimal parallel merging and sorting, *IEEE Trans. Comp.*, to appear.
- [2] B.-C. Huang and M.A. Langston, Stable duplicate-key extraction with optimal time and space bounds, *Acta Inform.* **26** (1989) 473-484.
- [3] B.-C. Huang and M.A. Langston, Fast stable merging and sorting in constant extra space, *Comput. J.*, to appear.
- [4] D.E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching* (Addison-Wesley, Reading, MA, 1973).
- [5] M.A. Kronrod, An optimal ordering algorithm without a field of operation, *Dokl. Akad. Nauk SSSR* **186** (1969) 1256-1258.
- [6] C.L. Liu, *Elements of Discrete Mathematics* (McGraw-Hill, New York, 1985).
- [7] E.M. Reingold, On the optimality of some set algorithms, *J. ACM* **19** (1972) 649-659.
- [8] L. Trabb Pardo, Stable sorting and merging with optimal space and time bounds, *SIAM J. Comput.* **6** (1977) 351-372.

<sup>2</sup> Because our procedures were devised with an eye toward clarity of exposition and not raw efficiency, and because our computations presuppose some sort of (possibly unrealizable) worst-case scenario, the figures we show are rather conservative upper bounds.

Reprinted from

# discrete applied mathematics

Discrete Applied Mathematics 54 (1994) 169–213

## Obstruction set isolation for the gate matrix layout problem

Nancy G. Kinnersley<sup>a,\*</sup>, Michael A. Langston<sup>b,2</sup>

<sup>a</sup>*Department of Computer Science, University of Kansas, Lawrence, KS 66045-2192, USA*

<sup>b</sup>*Department of Computer Science, University of Tennessee, Knoxville, TN 37996-1301, USA*

Received 9 January 1991; revised 4 December 1992



NORTH-HOLLAND AMSTERDAM



ELSEVIER

Discrete Applied Mathematics 54 (1994) 169–213

---

**DISCRETE  
APPLIED  
MATHEMATICS**

---

## Obstruction set isolation for the gate matrix layout problem

Nancy G. Kinnersley<sup>a,\*</sup>, Michael A. Langston<sup>b,2</sup><sup>a</sup>*Department of Computer Science, University of Kansas, Lawrence, KS 66045-2192, USA*<sup>b</sup>*Department of Computer Science, University of Tennessee, Knoxville, TN 37996-1301, USA*

Received 9 January 1991; revised 4 December 1992

---

### Abstract

Gate matrix layout is a well-known  $\mathcal{NP}$ -complete problem that arises at the heart of a number of VLSI layout styles. Despite its apparent general intractability, it has recently been shown that it can be solved in  $O(n^2)$  time whenever the number of tracks is fixed. Curiously, the proof of this is nonconstructive, based on finite but unknown obstruction sets. What then are such sets, and what is their underlying structure? The main result we report in this paper is a proof that the obstruction set for three tracks contains exactly 110 elements. We also describe a number of methods for obstruction identification that extend to any number of tracks.

*Key words:* Circuit layout; Finite-basis characterizations; Polynomial-time complexity

---

### 1. Introduction

Traditionally, decision problems<sup>3</sup> have been classified as either “easy” or “hard”, dependent on whether low-degree polynomial-time decision algorithms exist to solve them. Until recently, one could expect any proof of easiness to be *constructive*. That is, the proof itself should provide “positive evidence” in the form of the promised polynomial-time decision algorithm. Surprising advances, however, dramatically alter this appealing picture. See, for example, [6–8] for applications of tools from [14–17] that *nonconstructively* establish the existence of low-degree polynomial-time decision algorithms for a number of challenging combinatorial problems.

---

\*Corresponding author.

<sup>1</sup>This author's research has been supported in part by the University of Kansas under general research allocation 3800-20-0038 and by the National Science Foundation under grant CCR-9008725.

<sup>2</sup>This author's research has been supported in part by the National Science Foundation under grant MIP-8919312 and by the Office of Naval Research under contract N00014-90-J-1855.

<sup>3</sup>With its roots in set theory, computational complexity poses questions in terms of *decision* problems, rather than more natural *search* or *optimization* problems. Fortunately, the process of *self-reduction* often suffices to transform decision algorithms into search or optimization algorithms [3, 9, 10].

In general, problems amenable to this approach are modeled as graphs. The algorithm can decide whether a given encoding of a problem is a “yes” instance or a “no” instance by determining if it contains an element of a finite basis of forbidden graphs (the obstruction set). Strikingly, the underlying theory does not tell how to identify all members of such a set, the cardinality of the set, or even the order of the largest member of the set. The only fact we are given is that the set is finite.

Perhaps the best-known example of an algorithm based on such “negative evidence” is the celebrated finite-basis characterization of planar graphs [13]: a graph is planar if and only if it contains no member of a two-element obstruction set in the topological order. The main result we present in this paper is a similar finite-basis characterization for the three-track gate matrix layout problem: a graph represents a circuit with a three-track layout if and only if it contains no member of a 110-element obstruction set in the minor order.

Interestingly, it has recently been recognized [10] that gate matrix layout with parameter  $k$  is identical to the path-width problem with parameter  $k - 1$ . (That is, a graph  $G$  represents a circuit with a  $k$ -track layout if and only if  $G$  has a path decomposition [14] of width at most  $k - 1$ .) Because the work we report here was originally derived in terms of circuit layout, and because gate matrix layout has received considerable attention in the literature, we shall neither state nor prove our results in terms of path-width. Instead, we only note that it is fortuitous that our efforts contribute to the understanding of this important width metric.

Our proofs are of two general types. Some describe characteristics of obstructions, and thereby help to delimit the search space. Others show how a number of obstructions can be constructively obtained. Since these techniques alone are sufficient to bound but insufficient to isolate all obstructions, many obstructions were identified with the aid of exhaustive case-checking. To assist in this heroic undertaking, massive computational power<sup>4</sup> was used to verify that each obstruction represents a circuit that has no three-track layout, and to check that each proper minor of each obstruction represents a circuit that does have a three-track layout.

In the next two sections, we discuss relevant background information. In Section 4, we present the notation and terminology used throughout the remainder of this paper. In Sections 5 and 6, we prove several general results and constructions that hold for any number of tracks. In Section 7, we determine some specific properties required of three-track layouts and isolate all nonouterplanar obstructions. In Section 8, we enumerate the entire three-track obstruction set and prove that this set is complete. In the final two sections, we summarize our work and pose a few related open problems.

<sup>4</sup>We employed the dynamic programming formulation as given in [4] and as streamlined in [12]. The algorithm's consumption of both time and space was enormous; its use was generally restricted to instances of moderate size (graphs with no more than about twenty edges) on an IBM 3090-300E.

## 2. The minor order

A graph  $H$  is less than or equal to a graph  $G$  in the *minor order*, written  $H \leq_m G$ , if and only if a graph isomorphic to  $H$  can be obtained from  $G$  by a series of these two operations: taking a subgraph and contracting an arbitrary edge. A family  $F$  of graphs is said to be *closed* under the minor order if the facts that  $G$  is in  $F$  and that  $H \leq_m G$  together imply that  $H$  must be in  $F$ . The *obstruction set* for a family  $F$  of graphs is the set of graphs in the complement of  $F$  that are minimal in the minor order. Therefore, if  $F$  is closed under the minor order, it has the following characterization:  $G$  is in  $F$  if and only if  $H \not\leq_m G$  for every  $H$  in the obstruction set for  $F$ .

**Theorem 2.1** [17]. *Any set of finite graphs contains only a finite number of minor-minimal elements.*

**Theorem 2.2** [16]. *For every fixed graph  $H$ , the problem that takes as input a graph  $G$  and determines whether  $H \leq_m G$  is solvable in polynomial time.*

Theorems 2.1 and 2.2 guarantee only the existence of a polynomial-time decision algorithm for any minor-closed family of graphs. In particular, *no* proof of Theorem 2.1 can be entirely constructive [10].

Letting  $n$  denote the number of vertices in  $G$ , the time bound for algorithms ensured by these theorems is  $O(n^3)$ . If  $F$  excludes a planar graph, then the bound reduces to  $O(n^2)$ . In general, these algorithms possess enormous constants of proportionality [15], although new techniques greatly mitigate them [18], and methods specific to layout problems such as the one we address here lower them even more [10].

## 3. The gate matrix layout problem

Gate matrix layout is a combinatorial problem that arises in several VLSI layout styles, including gate matrix, PLAs under multiple folding, Weinberger arrays and others. It was originally posed in terms of operations on Boolean matrices. Formally, we are given an  $n \times m$  Boolean matrix  $M$  and an integer  $k$ , and are asked whether we can permute the columns of  $M$  so that, if in each row we change to \* every 0 lying between the row's leftmost and rightmost 1, then no column contains more than  $k$  1s and \*s. Such a \* is termed a *fill-in*. We refer the interested reader to [4] for sample instances, figures and additional background on this challenging problem.

Although the general problem is  $\mathcal{NP}$ -complete, it has been shown that, for any fixed value of  $k$ , an arbitrary instance can be mapped to an equivalent instance with only two 1s per column, then modeled as a graph on  $n$  vertices such that the family of “yes” instances is closed under the minor order and excludes a planar graph.

**Theorem 3.1** [6]. *For any fixed  $k$ , gate matrix layout can be decided in  $O(n^2)$  time.*



Fig. 1. Obstruction set for 2-GML.

Thanks to this mapping defined on arbitrary Boolean matrices, it suffices to restrict our attention to connected, simple graphs.

In the sequel, we shall use the term  $k$ -GML to denote the  $k$ -track variant of gate matrix layout. Thus, an obstruction for  $k$ -GML is a graph that represents a “no” instance for parameter  $k$  (it has no  $k$ -track layout) and that is minimal for parameter  $k$  (each of its proper minors does have a  $k$ -track layout). For 1-GML, it is trivial to see that the obstruction set contains only  $K_2$ . For 2-GML, the only obstructions are  $K_3$  and  $S(K_{1,3})$ <sup>5</sup> [2] (see Fig. 1). (The connected graphs that are “yes” instances for 2-GML are known as *caterpillars*.)

#### 4. Definitions and notation

Let  $G$  denote a graph, with vertex set  $V$  and edge set  $E$ , and let  $M$  denote an incidence matrix for  $G$ , augmented as necessary with fill-ins. For convenience, we assume a labeling for  $V$  and some appropriate bijection between these labels and the rows of  $M$ . Thus we shall, for example, refer merely to “row  $u$ ” rather than to the more precise but cumbersome “row that corresponds to vertex  $u$ ”.

We term the matrix  $M$  a *permutation* for  $G$ , since the ordering of its columns determines an ordering for  $E$ . The *cost of a column* is the total number of 1s and fill-ins it contains. The *cost of a permutation* is the maximum cost of any of its columns. The *cost of a graph* is the minimum cost of any of its permutations. These costs represent the number of tracks required in a layout of the associated circuit.

A vertex of degree one is a *pendant vertex*. A (simple) *path* is a sequence of distinct vertices  $v_1, v_2, \dots, v_h$  such that edge  $v_i v_{i+1} \in E$  for  $1 \leq i < h$ . Vertices that form such a sequence are *consecutive*. A *pendant path* is a path in which  $v_1$  has degree three or more,  $v_h$  has degree one, and each  $v_i$ ,  $1 < i < h$ , has degree two. The *length* of such a path is  $h - 1$ , the number of edges it contains.

A planar graph along with a planar embedding is called a *plane graph*. Similarly, an outerplanar graph [11] along with an outerplanar embedding is called an *outerplane graph*. The regions of the plane bounded by the embedding are called *faces*. (The unbounded region is known as the “exterior” face. Unless otherwise noted, a face is understood to mean an interior face.) Two faces in a plane graph are *edge adjacent* if

<sup>5</sup> $S(K_{1,3})$  is the graph obtained by subdividing each edge of  $K_{1,3}$ .

their intersection contains one or more edges. Two faces are *vertex adjacent* if their intersection contains one or more vertices but no edges.

Given a permutation for a graph, the *span for a vertex* is the collection of columns that contain either a 1 or a fill-in in its row. If the graph is plane, then the *span for a face* is the collection of columns that lie between the leftmost and rightmost columns that represent edges of the face, inclusive.

Finally, we assume the reader is familiar with standard graph operations, in particular subtraction ( $\setminus$ ), union ( $\cup$ ) and intersection ( $\cap$ ) [1].

## 5. Obstruction characterization tools

In this section and the next, we shall derive<sup>6</sup> a number of results that help to characterize or construct obstructions. These results hold for arbitrary  $k$ .

**Lemma 5.1.** *No obstruction for  $k$ -GML contains two or more pendant paths of length one incident on a common vertex.*

**Proof.** Assume otherwise, and let  $G$  denote an obstruction for  $k$ -GML with pendant vertices  $v$  and  $w$ , both adjacent to vertex  $u$ . Let  $G' = G \setminus \{w\}$ . Since  $G$  is minimal for parameter  $k$ ,  $G'$  possesses a permutation  $M'$  with cost at most  $k$ . (Recall that permutations are augmented only as necessary with fill-ins, and so  $M'$  has no fill-ins whatsoever in row  $v$ .) Consider the matrix  $M$  obtained from  $M'$  by adding row  $w$  and placing column  $uw$  adjacent to column  $uv$ . The cost of column  $uw$  is identical to that of column  $uv$ , because still no fill-ins are needed in row  $v$ . Moreover, the costs of all other columns remain unchanged, because no fill-ins are required anywhere in row  $w$ . Therefore,  $M$  is a permutation for  $G$  with cost at most  $k$ , contradicting our assumption that  $G$  has no  $k$ -track layout.  $\square$

**Lemma 5.2.** *No obstruction for  $k$ -GML contains a pendant path of length greater than two.*

**Proof.** Assume otherwise, and let  $G$  denote an obstruction for  $k$ -GML with pendant path  $x, \dots, u, v, w$  of length three or more. Let  $G' = G \setminus \{w\}$ . Because  $G$  is minimal and because  $G' <_m G$ , there is an optimal permutation  $M'$  for  $G'$  with cost at most  $k$ . Since  $u$  has degree two, we may assume by symmetry that column  $uv$  contains the rightmost 1 in row  $u$ . Consider the matrix  $M$  obtained from  $M'$  by adding row  $w$  and inserting column  $vw$  to the immediate right of column  $uv$ . Since column  $vw$  does not need

<sup>6</sup>As a matter of style, we shall omit proofs when they follow immediately from previous results, and shall merely point the reader to an earlier proof when analogous arguments suffice. We realize that the responsibility for deciding when to suppress details in this presentation is ours alone, and remark that full proofs, even of the corollaries, can be found in [12].

a fill-in in row  $u$ , its cost is the same as that of column  $uv$ . The costs of all other columns remain unchanged, because no fill-ins are required in row  $v$  or in row  $w$ . Therefore,  $M$  is a permutation for  $G$  with cost at most  $k$ , contradicting the assumption that  $G$  has no  $k$ -track layout.  $\square$

Thus, a pendant vertex is an endpoint of either a pendant path of length one (which we shall henceforth call a *pendant edge*) or a pendant path of length two (which we shall without ambiguity henceforth term simply a *pendant path*, omitting reference to its length).

**Lemma 5.3.** *If a graph has a pendant path, then there is an optimal permutation for that graph in which the edges of the path are represented by adjacent columns.*

**Proof.** Let  $G$  denote a graph with pendant path  $u, v, w$ , and let  $M$  denote an optimal permutation for  $G$ . Suppose that columns  $uv$  and  $vw$  are not adjacent, and that column  $uv$  is to the left of column  $vw$ . The rightmost 1 in row  $u$  must be either (1) in column  $uv$ , (2) in a column between columns  $uv$  and  $vw$ , or (3) in a column to the right of column  $vw$ .

Suppose (1) holds. We construct a new matrix  $M'$  from  $M$  by moving column  $vw$  to the left until it is to the immediate right of column  $uv$ . Since column  $vw$  does not require a fill-in in row  $u$ , its cost is no more than that of column  $uv$ . Moreover, no column now requires a fill-in in row  $v$ , and the cost of  $M'$  is no more than that of  $M$ .

Suppose (2) holds. For the sake of discussion, assume that the rightmost 1 in row  $u$  is in column  $ux$ . We construct matrix  $M'$  from  $M$  by first moving column  $uv$  to the right until it is to the immediate right of column  $ux$ . Since column  $ux$  had a fill-in in row  $v$ , the cost of column  $uv$  is no more than the original cost of column  $ux$ . To complete the construction of  $M'$ , move column  $vw$  to the left until it is to the immediate right of column  $uv$ . Since column  $vw$  does not require a fill-in in row  $u$ , its cost is no more than that of column  $uv$ . Therefore, the cost of  $M'$  is no greater than that of  $M$ .

Suppose (3) holds. We construct matrix  $M'$  from  $M$  by moving column  $uv$  to the right until it is to the immediate left of column  $vw$ . Since column  $vw$  has a fill-in in row  $u$ , the cost of column  $uv$  is no more than the cost of column  $vw$ . Thus, the cost of  $M'$  cannot exceed that of  $M$ .  $\square$

**Lemma 5.4.** *No obstruction for  $k$ -GML contains more than three pendant paths incident on a common vertex.*

**Proof.** Assume otherwise, and let  $G$  denote an obstruction for  $k$ -GML with four or more pendant paths incident on vertex  $u$ . Let  $u, v, w$  be one such pendant path, and let  $G' = G \setminus \{uv, vw\}$ . Because  $G$  is minimal and because  $G' \leq_m G$ ,  $G'$  possesses a permutation  $M'$  with cost at most  $k$  in which (due to Lemma 5.3) each pendant path incident on  $u$  is represented by a pair of adjacent columns. Let the second such pair of columns represent pendant path  $u, x, y$ . (We choose the second pair of columns since this

guarantees that column  $xy$  has a fill-in in row  $u$ .) We construct matrix  $M$  from  $M'$  by adding rows  $v$  and  $w$ , and by placing columns  $uv$  and  $vw$  to the immediate right of columns  $ux$  and  $xy$ . Since no fill-ins are required in rows  $x$  and  $y$ , the costs of columns  $uv$  and  $vw$  are the same as the costs of columns  $ux$  and  $xy$ , respectively. Therefore,  $M$  is a permutation for  $G$  with cost at most  $k$ , contradicting our assumption that  $G$  has no  $k$ -track layout.  $\square$

**Lemma 5.5.** *For  $k > 2$ , no obstruction for  $k$ -GML contains more than two consecutive vertices of degree two.*

**Proof.** Assume otherwise, and let  $G$  denote an obstruction for  $k$ -GML,  $k > 2$ , with consecutive vertices  $u, v$  and  $w$ , each of degree two. Let  $G'$  be the graph obtained from  $G$  by contracting the edge  $uv$  to  $u$ . (Observe that  $u$  and  $w$  each retain degree two in  $G'$ : no increase in degree is possible; a decrease would imply either that  $G$  is  $K_3$  and hence not an obstruction for  $k > 2$ , or that  $G$  is not connected and hence not an obstruction for any  $k$ .) Because  $G$  is minimal and because  $G' \leq_m G$ ,  $G'$  must possess an optimal permutation  $M'$  with cost at most  $k$ . From the facts that  $M'$  has no unnecessary fill-ins in rows  $u$  and  $w$  and that both  $u$  and  $w$  have degree two, it follows that either (1) the spans for these two rows overlap only in column  $uw$  or (2) the span for one properly contains the span for the other.

Suppose (1) holds. For the sake of discussion, assume the single column of overlap (column  $uw$ ) contains the rightmost 1 in row  $u$  and thus the leftmost 1 in row  $w$ . We construct at no extra cost a new matrix  $M$  from  $M'$ , by replacing column  $uw$  with columns  $uv$  and  $vw$ .

Suppose (2) holds. For the sake of discussion, assume the span for  $u$  properly contains the span for  $w$ , with column  $uw$  the rightmost in both spans. Let  $c$  denote the column that contains the leftmost 1 in row  $w$ . We construct at no extra cost a new matrix  $M$  from  $M'$ , by replacing column  $uw$  with column  $vw$ , and by inserting column  $uv$  to the immediate left of column  $c$ .

In either case,  $M$  is a permutation for  $G$  with cost at most  $k$ , contradicting the assumption that  $G$  has no  $k$ -track layout.  $\square$

**Lemma 5.6.** *Suppose  $G$  contains a pair of adjacent vertices,  $u$  and  $v$ , each of degree two. If  $G'$  is obtained from  $G$  by contracting the edge  $uv$  to  $u$ , adding a new vertex  $w$ , and adding the edge  $uw$ , then the cost of  $G$  equals that of  $G'$ .*

**Proof.** Let  $G, G', u, v$  and  $w$  be as defined in the statement of the lemma. Let  $t(x)$  denote the other vertex adjacent to  $u$  ( $v$ ) in  $G$ . Let  $M$  denote an optimal permutation for  $G$ .

We construct a new matrix  $M'$  from  $M$  by replacing column  $vx$  with column  $ux$  and changing the label on row  $v$  to  $w$ . Row  $w$  contains a single 1 and requires no fill-ins. Any column that now needs a new fill-in in row  $u$  originally had a fill-in in row  $v$ . Thus, the cost of  $M'$  is no more than that of  $M$ . Because  $M'$  is a permutation for  $G'$ , the cost of  $G'$  cannot exceed that of  $G$ .

Let  $M'$  denote an optimal permutation for  $G'$ . Note that, in  $G'$ , vertex  $u$  has degree three and is adjacent to vertices  $t$ ,  $w$ , and  $x$ . It suffices to consider two cases for  $M'$ , in that either (1) column  $uw$  lies between columns  $tu$  and  $ux$  or (2) column  $uw$  contains the leftmost 1 in row  $u$ .

Suppose (1) holds. We construct a new matrix  $M$  from  $M'$  by replacing column  $ux$  with column  $wx$  and changing the label on row  $w$  to  $v$ . Any fill-ins required in row  $v$  lie in columns that no longer require fill-ins in row  $u$ . Thus, the cost of  $M$  is no more than that of  $M'$ .

Suppose (2) holds. If column  $uw$  has a fill-in in row  $t$  (or row  $x$ ) then, at no extra cost, we move column  $tu$  (column  $ux$ ) to the immediate left of column  $uw$ .  $M$  can now be constructed as in (1). If column  $uw$  has 0s in both row  $t$  and row  $x$ , then we may assume that columns  $tu$  and  $ux$  contain the leftmost 1s in rows  $t$  and  $x$ , respectively. (To see this, note that if another column  $c$  holds the leftmost 1 in row  $t$  (row  $x$ ), then  $c$  has a fill-in in row  $u$  and column  $tu$  (column  $ux$ ) can be placed to the left of  $c$  with no increase in cost). If column  $ux$  is to the right of column  $tu$  then, at no extra cost, we move column  $uw$  to the immediate left of column  $ux$ . Otherwise, at no extra cost, we move column  $uw$  to the immediate left of column  $tu$ .  $M$  can now be constructed as in (1).

Because  $M$  is a permutation for  $G$ , the cost of  $G$  cannot exceed that of  $G'$ .  $\square$

**Corollary 5.7.** *If  $G$  and  $G'$  denote graphs as defined in Lemma 5.6, then  $G$  is an obstruction for  $k$ -GML if and only if  $G'$  is.*

**Corollary 5.8.** *No obstruction for  $k$ -GML contains two adjacent vertices of degree three each adjacent to a pendant vertex as well.*

**Corollary 5.9.** *No obstruction for  $k$ -GML contains a vertex of degree three adjacent to both a pendant vertex and a vertex of degree two.*

**Lemma 5.10.** *Let  $G$  denote an arbitrary graph with cost  $k$ , and let  $v$  denote any vertex of  $G$ .  $G$  possesses an optimal permutation in which every column with cost  $k$  has a nonzero entry in row  $v$  if and only if  $G \setminus \{v\}$  does not contain an obstruction for  $(k - 1)$ -GML.*

**Proof.** Let  $G$  denote a graph with cost  $k$  and let  $v$  denote any vertex of  $G$ . If  $G \setminus \{v\}$  contains an obstruction for  $(k - 1)$ -GML, then every optimal permutation for  $G \setminus \{v\}$  has cost  $k$ . Therefore, every optimal permutation for  $G$  contains a column with cost  $k$  that has a 0 in row  $v$ .

If  $G \setminus \{v\}$  does not contain an obstruction for  $(k - 1)$ -GML, then  $G \setminus \{v\}$  possesses an optimal permutation  $M'$  with cost at most  $k - 1$ . Consider the matrix  $M$  obtained from  $M'$  by adding row  $v$  and, for each vertex  $w$  adjacent to  $v$  in  $G$ , inserting column  $vw$  adjacent to any column with a 1 in row  $w$ . In every case, the cost of column  $vw$  is at most  $k$ . Since  $v$  is the only row that may need additional fill-ins,  $M$  is an optimal permutation for  $G$  of cost  $k$  in which every column with cost  $k$  has a nonzero entry in row  $v$ .  $\square$

**Corollary 5.11.** *Let  $G$  denote an obstruction for  $k$ -GML and let  $v$  denote a vertex of  $G$ .  $G$  has cost exactly  $k + 1$ , and possesses an optimal permutation in which every column of cost  $k + 1$  has a 1 or a fill-in in row  $v$ .*

**Lemma 5.12.** *Adding an edge to a graph increases its cost by at most one.*

**Proof.** Straightforward.  $\square$

**Lemma 5.13.** *If  $G$  contains  $K_k$  as a subgraph, then  $G$  has cost at least  $k$  and possesses an optimal permutation in which the edges of  $K_k$  are represented in adjacent columns.*

**Proof.** Follows immediately from Lemma 4.1 of [6].  $\square$

**Corollary 5.14.** *If  $G$  contains  $K_k$  as a minor, then  $G$  has cost at least  $k$ .*

**Corollary 5.15.**  *$K_k$  is an obstruction for  $(k - 1)$ -GML.*

**Lemma 5.16.** *Let  $G$  denote an arbitrary graph and let  $H_1$  and  $H_2$  denote obstructions for  $k$ -GML. If  $G \cap H_1 = G \cap H_2 = \{v\}$  for some vertex  $v$ , then the cost of  $G \cup H_1$  equals that of  $G \cup H_2$ .*

**Proof.** Let  $G, H_1, H_2$  and  $v$  be as defined in the statement of the lemma. Let  $M$  denote an optimal permutation for  $G \cup H_1$ .

Since  $H_1$  is an obstruction for  $k$ -GML, some column  $c$  of  $H_1$  in  $M$  has cost  $k + 1$  in the rows of  $H_1$ . Either  $c$  is contained in the span for  $v$  (in which case  $c$  contains a 1 or a fill-in in row  $v$ ), or else the connectedness of  $H_1$  ensures that every column of  $G$  lying between  $c$  and the span for  $v$  has a fill-in in some other row of  $H_1$ .

Due to Corollary 5.11,  $H_2$  possesses a cost  $k + 1$  permutation  $M_2$  in which every column with cost  $k + 1$  has a 1 or a fill-in in row  $v$ . We use  $M_2$  to construct a new matrix  $M'$  from  $M$  as follows. We first eliminate the rows of  $H_1 \setminus \{v\}$ , then all resultant columns with at most one 1 (one of which is  $c$ ). We next insert  $M_2$  into the position formerly occupied by  $c$  (which requires a new row for each vertex of  $H_2 \setminus \{v\}$ ). No inserted column can require more fill-ins than did  $c$ . Due to the way  $c$  was chosen and its relation to the span for  $v$ , no column originally in  $M$  can incur an increase in its number of fill-ins. Thus, the cost of  $M'$  is no more than that of  $M$ .

Because  $M'$  is a permutation for  $G \cup H_2$ , the cost of  $G \cup H_2$  cannot exceed that of  $G \cup H_1$ . The inequality is established in the reverse direction analogously.  $\square$

**Corollary 5.17.** *If  $G, H_1$  and  $H_2$  denote graphs as defined in Lemma 5.16, and if  $G \cup H_1$  is an obstruction for  $k'$ -GML but  $G \cup H_2$  is not, then any obstruction for  $k'$ -GML contained as a minor in  $G \cup H_2$  has the form  $G \cup H'_2$  for some  $H'_2 <_m H_2$ .*

**Lemma 5.18.** *Let  $G$  be a plane graph with face  $F$ . In any permutation for  $G$ , every column in the face span for  $F$  has a cost of at least two in the collection of rows that*

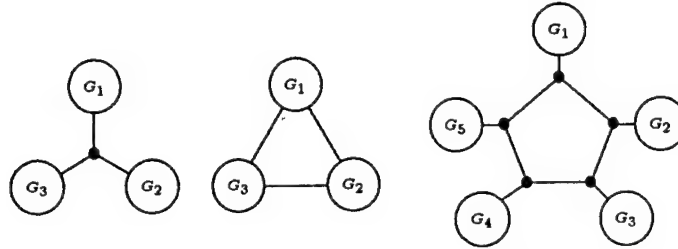


Fig. 2. Constructions used in Lemmas 6.1, 6.2 and 6.4.

correspond to the vertices of  $F$ , and every interior column of that span has a total cost of at least three.

**Proof.** Straightforward.  $\square$

## 6. Obstruction construction tools

The constructions studied in this section are depicted informally in Fig. 2.

**Lemma 6.1.** Let  $G_1, G_2$  and  $G_3$  denote disjoint (but not necessarily distinct) obstructions for  $k$ -GML, let  $v_i$  denote an arbitrary vertex of  $G_i$  for  $1 \leq i \leq 3$ , and let  $v$  denote an isolated vertex not in  $G_1 \cup G_2 \cup G_3$ . The graph  $G = G_1 \cup G_2 \cup G_3 \cup \{v\} \cup \{vv_i : 1 \leq i \leq 3\}$  is an obstruction for  $(k+1)$ -GML.

**Proof.** Let  $G_i, v_i, v$  and  $G$  be as defined in the statement of the lemma. It follows from Lemma 4.3 of [6] that  $G$  has cost  $k+2$ .

We now establish the minimality of  $G$ . Due to Corollary 5.11, each  $G_i$ ,  $1 \leq i \leq 3$  possesses a cost  $k+1$  permutation  $M_i$  in which every column with cost  $k+1$  has a 1 or a fill-in in row  $v_i$ . Since  $G$  is connected, the removal of a vertex necessarily means the removal of an edge and, therefore, we only need consider the effect of removing or contracting a single edge,  $e$ . Because of  $G$ 's symmetry, we may assume that either (1)  $e$  is in  $G_1$ , or (2)  $e = vv_1$ .

Suppose (1) holds. Let  $G'_1$  and  $G'$  denote the minors of  $G_1$  and  $G$ , respectively, that are obtained by the removal or contraction of  $e$  (for notational simplicity in the case of a contraction, we insist that  $e$  be contracted to  $v_1$  if  $v_1 \in e$ ). Because  $G_1$  is minimal for parameter  $k$ ,  $G'_1$  possesses a permutation  $M'_1$  with cost at most  $k$ . Let  $M$  denote the permutation  $M_2, vv_2, M'_1, vv_3, M_3$ . A column in  $M_2$  or  $M_3$  has cost at most  $k+1$ . Columns  $vv_2$  and  $vv_3$  each have cost two. Any column in  $M'_1$  incurs one additional fill-in (in row  $v$ ), bringing its cost to at most  $k+1$ . Thus,  $M$  has cost  $k+1$ . We form  $M'$  from  $M$  at no extra cost by placing  $vv_1$  adjacent to an arbitrary column in  $M$  with a 1 in row  $v_1$ .

Suppose (2) holds. If  $G' = G \setminus \{e\}$ , let  $M'$  denote the permutation  $M_2, vv_2, vv_3, M_3, M_1$ . Since  $v$  now has degree two, no fill-ins are required in its row. Because of the way  $M_2$  and  $M_3$  were chosen, any column that requires a new fill-in in row  $v_2$  or row  $v_3$  has cost at most  $k + 1$ , and so  $M'$  has cost  $k + 1$ . If  $G'$  is obtained from  $G$  by contracting  $e$  to  $v_1$ , let  $M'$  denote the permutation  $M_2, v_1v_2, M_1, v_1v_3, M_3$ . Only the  $v_i$  rows may require additional fill-ins. Again, because of the way each  $M_i$  was chosen, any new fill-in must lie in a column that has cost at most  $k + 1$ , and so  $M'$  has cost  $k + 1$ .

In any case,  $M'$  is a permutation for  $G'$ , and thus the cost of  $G'$  is strictly less than that of  $G$ .  $\square$

**Lemma 6.2.** Let  $G_1, G_2$ , and  $G_3$  denote disjoint (but not necessarily distinct) graphs of cost  $k$ , and let  $u_i$  and  $v_i$  denote arbitrary (but not necessarily distinct) vertices of  $G_i$  for  $1 \leq i \leq 3$ . The graph  $G = G_1 \cup G_2 \cup G_3 \cup \{v_1v_2, u_1v_3, u_2u_3\}$  has cost at least  $k + 1$ .

**Proof.** Let  $G_i, u_i, v_i$  and  $G$  be as defined in the statement of the lemma. Let  $M$  denote an optimal permutation for  $G$  and, in  $M$ , let  $c_i$  denote a column of  $G_i$  with cost at least  $k$  in the rows of  $G_i$ . Without loss of generality, assume  $c_1$  lies to the left of  $c_2$  which lies to the left of  $c_3$ . If  $u_1v_3$  lies to the left of  $c_2$ , then  $c_2$  has a fill-in in some row of  $G_3$ . Otherwise, it has a fill-in in some row of  $G_1$ . Thus the cost of  $G$  is at least  $k + 1$ .  $\square$

The graph  $G$  just defined may not, however, have cost exactly  $k + 1$ , even if  $u_i = v_i$ ,  $1 \leq i \leq 3$ . An example is illustrated in Fig. 3.

**Corollary 6.3.** Let  $G_1, G_2$ , and  $G_3$  denote obstructions for  $(k - 1)$ -GML, and let  $v_i$  denote an arbitrary vertex of  $G_i$  for  $1 \leq i \leq 3$ . The graph  $G = G_1 \cup G_2 \cup G_3 \cup \{v_1v_2, v_1v_3, v_2v_3\}$  has cost exactly  $k + 1$ .

The graph  $G$  just defined may not, however, be an obstruction for  $k$ -GML. For example, obstruction 12.3.1 listed in the appendix is properly contained in the graph constructed by setting  $G_1 = G_2 = K_3$  and setting  $G_3 = S(K_{1,3})$ , with  $v_3$  a vertex of degree one.

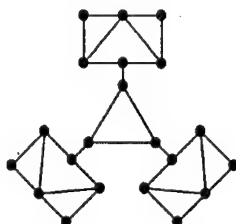


Fig. 3. A graph of cost 5 built from three graphs of cost 3.

**Lemma 6.4.** Let  $G_1, G_2, G_3, G_4$  and  $G_5$  denote disjoint (but not necessarily distinct) obstructions for  $k$ -GML, and let  $u_i$  denote an arbitrary vertex of  $G_i$  for  $1 \leq i \leq 5$ . Let  $C_5$  denote a cycle graph of order five, with vertex set  $\{v_i: 1 \leq i \leq 5\}$ , disjoint from  $G_1 \cup G_2 \cup G_3 \cup G_4 \cup G_5$ . The graph  $G = G_1 \cup G_2 \cup G_3 \cup G_4 \cup G_5 \cup \{u_i v_i: 1 \leq i \leq 5\}$  is an obstruction for  $(k + 2)$ -GML.

**Proof.** Let  $G_i, u_i, C_5, v_i$  and  $G$  be as defined in the statement of the lemma. Let  $M$  denote an optimal permutation for  $G$  and, in  $M$ , let  $c_i$  denote a column of  $G_i$  with cost at least  $k + 1$  in the rows of  $G_i$ . If any  $c_i$  lies between the leftmost and rightmost columns of  $C_5$ , then it incurs at least two additional fill-ins (in rows of  $C_5$ ). Otherwise, without loss of generality, assume  $c_1$  lies to the left of  $c_2$  which lies to the left of  $c_3$  which lies to the left of the leftmost column of  $C_5$ . In this event,  $c_3$  incurs two additional fill-ins (one in a row of  $G_1 \cup \{v_1\}$ , and one in a row of  $G_2 \cup \{v_2\}$ ). Thus the cost of  $G$  is at least  $k + 3$ .

Letting  $M_i$  denote a cost  $k + 1$  permutation for  $G_i$  in which every column with cost  $k + 1$  has a 1 or a fill-in in row  $u_i$ , we observe that  $G$  has cost exactly  $k + 3$  as evidenced by the permutation  $M_1, u_1 v_1, M_2, u_2 v_2, v_1 v_2, v_2 v_3, u_3 v_3, M_3, v_3 v_4, v_4 v_5, v_1 v_5, u_4 v_4, M_4, u_5 v_5, M_5$ .

We now establish the minimality of  $G$ . As in the proof of Lemma 6.1, we need only consider the effect of removing or contracting a single edge,  $e$ , and may assume that either (1)  $e$  is in  $G_1$ , (2)  $e = u_1 v_1$ , or (3)  $e = v_1 v_2$ .

Suppose (1) holds. Let  $G'_1$  and  $G'$  denote the minors of  $G_1$  and  $G$ , respectively, that are obtained by the removal or contraction of  $e$  (if a contraction,  $e$  is contracted to  $u_1$  if  $u_1 \in e$ ). Because  $G_1$  is minimal for parameter  $k$ ,  $G'_1$  possesses a permutation  $M'_1$  with cost at most  $k$ . Let  $M'_1$  denote the matrix formed at no extra cost from  $M'_1$  by adding row  $v_1$  and placing column  $u_1 v_1$  adjacent to an arbitrary column with a 1 in row  $u_1$ . Let  $M'$  denote the permutation  $M_2, u_2 v_2, M_3, u_3 v_3, v_2 v_3, v_1 v_2, M'_1, v_3 v_4, v_1 v_5, v_4 v_5, u_4 v_4, M_4, u_5 v_5, M_5$ , which has cost at most  $k + 2$ .

Suppose (2) holds. If  $G' = G \setminus \{e\}$ , let  $M'$  denote the permutation  $M_2, u_2 v_2, M_3, u_3 v_3, v_2 v_3, v_1 v_2, v_3 v_4, v_4 v_5, v_1 v_5, u_4 v_4, M_4, u_5 v_5, M_5, M_1$ , which has cost at most  $k + 2$ . If  $G'$  is obtained from  $G$  by contracting  $e$  to  $u_1$ , let  $M'$  denote the permutation  $M_2, u_2 v_2, M_3, u_3 v_3, v_2 v_3, v_2 u_1, M_1, u_1 v_5, v_4 v_5, v_3 v_4, u_4 v_4, M_4, u_5 v_5, M_5$ , which has cost at most  $k + 2$ .

Suppose (3) holds. If  $G' = G \setminus \{e\}$ , let  $M'$  denote the permutation  $M_2, u_2 v_2, v_2 v_3, u_3 v_3, M_3, v_3 v_4, u_4 v_4, M_4, v_4 v_5, u_5 v_5, M_5, v_5 v_1, u_1 v_1, M_1$ , which has cost at most  $k + 2$ . If  $G'$  is obtained from  $G$  by contracting  $e$  to  $v_1$ , let  $M'$  denote the permutation  $M_1, u_1 v_1, M_2, u_2 v_1, M_3, u_3 v_3, v_1 v_3, v_3 v_4, v_1 v_5, v_4 v_5, u_4 v_4, M_4, u_5 v_5, M_5$ , which has cost at most  $k + 2$ .

In any case,  $M'$  is a permutation for  $G'$ , and thus the cost of  $G'$  is strictly less than that of  $G$ .  $\square$

## 7. Special tools for three-track obstructions

Unlike the work of the last two sections, the results we now derive hold only for  $k = 3$ .

### 7.1. General properties of the three-track obstructions

**Lemma 7.1.** *No obstruction for 3-GML contains a vertex of degree four or more adjacent to a pendant vertex.*

**Proof.** Assume otherwise, and let  $G$  denote an obstruction for 3-GML with vertex  $v$  adjacent to vertices  $w, x, y$  and pendant vertex  $z$ . Let  $G' = G \setminus \{z\}$ , and let  $M'$  denote a cost-three permutation for  $G'$ . Without loss of generality, assume that column  $vw$  lies to the left of both  $vx$  and  $vy$ , and that column  $vw$  has a 0 in row  $x$ . Let  $c$  denote the column that contains the leftmost 1 in row  $x$ . We construct matrix  $M$  from  $M'$  by adding row  $z$  and placing column  $vz$  to the immediate left of  $c$ .  $M$  is a permutation for  $G$  with cost at most three, contradicting our assumption that  $G$  has no three-track layout.  $\square$

This result (aided by the corollaries to Lemma 5.6) is easily extended.

**Corollary 7.2.** *No obstruction for 3-GML contains two adjacent vertices each adjacent to a pendant vertex.*

Given a permutation for a plane graph, the *overlap* of two or more face spans is the collection of columns common to all spans.

**Lemma 7.3.** *If a plane graph of cost three contains two faces whose intersection is exactly one vertex, then it possesses an optimal permutation in which the overlap of the spans for these faces is empty.*

**Proof.** Let  $G$  denote a plane graph of cost three with faces  $F_1$  and  $F_2$  such that  $F_1 \cap F_2 = v$ . Let  $M$  denote a cost-three permutation for  $G$ , and suppose the overlap of the face spans for  $F_1$  and  $F_2$  is nonempty. Because these faces are not edge adjacent, their overlap contains at least two columns, each with cost two in the rows of  $F_1$ , and each with cost two in the rows of  $F_2$  (Lemma 5.18). Since  $M$  has cost three, and since  $v$  is the only vertex on both  $F_1$  and  $F_2$ , each column of the overlap represents an edge of either  $F_1$  or  $F_2$  that is incident on  $v$ .

Without loss of generality, assume the leftmost column of the overlap is  $vw$  of  $F_2$ , with a fill-in in row  $u$  of  $F_1$ . Since the cost of  $M$  is three, the column to the immediate right of  $vw$  must be  $uv$ . If  $vx$  of  $F_1$  is to the right of  $uv$ , then  $vw$  requires a fill-in in row  $x$  as well, contradicting the fact that  $M$  has cost three. Therefore, the overlap contains only  $vw$  and  $uv$ , and interchanging the two columns yields a cost-three permutation for  $G$  with the desired property.  $\square$

**Lemma 7.4.** *If a plane graph of cost three contains two faces whose intersection is exactly one edge, then it possesses an optimal permutation in which the overlap of the spans for these faces is exactly one column.*

**Proof.** Let  $G$  denote a plane graph of cost three with faces  $F_1$  and  $F_2$  such that  $F_1 \cap F_2 = uv$ . Given a cost-three permutation for  $G$ , suppose the overlap of the face spans for  $F_1$  and  $F_2$  contains two or more columns (it cannot be empty because it must contain  $uv$ ). Moreover, suppose the overlap contains no pendant edges incident on  $u$  or  $v$  (any such edge can be removed initially, then reinserted after our forthcoming permutation modification at no extra cost).

Without loss of generality, assume that the rightmost column of  $F_1$  lies to the right of both  $uv$  and the leftmost column of  $F_2$ . It is straightforward to verify that the overlap contains at most three columns, that  $uv$  and the leftmost column of  $F_2$  are the same, and that the column to the immediate right of  $uv$  must have the form  $uw$  (or  $vw$ ) for some  $w \in F_1$ . Thus column  $uv$  must have a fill-in in row  $w$ ,  $uw$  (or  $vw$ ) must have a fill-in in row  $v$  (or  $u$ ), and so  $uv$  and  $uw$  (or  $vw$ ) can be interchanged at no extra cost, an action which eliminates a column from the overlap. At most one more application of this interchange reduces the overlap to  $uv$  alone.  $\square$

## 7.2. Three-track obstructions that are not outerplanar

Since  $K_4$ , an obstruction for 3-GML, is a minor of both  $K_5$  and  $K_{3,3}$ , all obstructions for 3-GML are planar. We now establish that  $K_4$  and the four graphs illustrated in Fig. 4 are the only obstructions for 3-GML that are not outerplanar.

**Lemma 7.5.** *The four graphs depicted in Fig. 4 are the only obstructions for 3-GML with the property that, for any planar embedding, there exists an edge not adjacent to the exterior face.*

**Proof.** Computation suffices to check that these four graphs are indeed obstructions for 3-GML; clearly, each has the property stated in the lemma. Thus we need only to establish that these are the only obstructions for 3-GML that possess this property.

Let  $G = \langle V, E \rangle$  denote an arbitrary plane obstruction for 3-GML with the desired property, and assume without loss of generality that its embedding maximizes the number of edges on or adjacent to the exterior face. Let  $V_f$  denote the set of vertices on this exterior face, and let  $V_n$  denote  $V \setminus V_f$ . Let  $G'$  denote the subgraph of  $G$  induced by  $V_n$ . Thus  $G'$  contains at least one edge,  $uv$ .

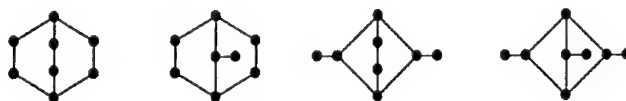


Fig. 4. Four related obstructions.

Let  $S$  denote the set of (simple) paths in  $G$  with an initial vertex in  $\{u, v\}$ , internal vertices in  $V_n$ , and a terminal vertex in  $V_f$ . If three or more distinct terminals are contained in the elements of  $S$ , then  $G \succ_m K_4$ , contradicting the presumed minimality of  $G$ . If every element of  $S$  contains the same terminal, then the connected component of  $G'$  containing  $uv$  can be moved to the exterior face, contradicting the presumed maximality of (the number of edges on or adjacent to) that face. Thus the elements of  $S$  contain exactly two different terminals, which we denote by  $w$  and  $x$ .

It now follows that  $G$  contains three vertex-disjoint paths from  $w$  to  $x$ . Moreover, the maximality of the exterior face dictates that each path either has length at least three, or contains an internal vertex adjacent to a distinct, additional vertex not on any of the three paths. Therefore  $G \geq_m H$  for some  $H$  depicted in Fig. 4.  $\square$

**Lemma 7.6.** *No obstruction for 3-GML contains a vertex of degree two adjacent to vertices of degree three or more unless those vertices are also adjacent.*

**Proof.** Assume otherwise, and let  $G$  denote a plane obstruction for 3-GML with degree two vertex  $v$  adjacent to vertices  $u$  and  $w$ , each of degree three or more, but not adjacent to each other. Lemma 7.1 and Corollary 5.9 guarantee that neither  $u$  nor  $w$  is adjacent to a pendant vertex. Let  $G'$  denote the minor of  $G$  obtained by contracting edge  $uv$  to  $u$ , and let  $M'$  denote a cost-three permutation for  $G'$ . Consider the overlap of the spans for  $u$  and  $w$ , and without loss of generality, assume the leftmost column is  $uw$  and that it contains the leftmost 1 in row  $w$ . If the overlap is  $uw$ , or if  $uw$  has cost two, adding row  $v$  and replacing  $uw$  with  $uv$  and  $vw$  produces a cost-three permutation for  $G$ , a contradiction. If  $uw$  has a fill-in in row  $x$ , it is straightforward to verify that some column of the overlap contains the rightmost 1 in row  $x$ , or that the overlap contains at most three columns one of which is  $ux$ . In either case, a cost-three permutation for  $G$  can be constructed from  $M'$ , again contradicting the assumption that  $G$  has no three-track layout. Therefore, an obstruction for 3-GML contains a vertex of degree two adjacent to vertices of degree three or more, only if (as obstruction 6.4.1 in the appendix illustrates) the three vertices are pairwise adjacent.  $\square$

**Lemma 7.7.**  *$K_4$  and the graphs depicted in Fig. 4 are the only obstructions for 3-GML that are not outerplanar.*

**Proof.** Assume otherwise. Let  $G$  denote a nonouterplanar obstruction for 3-GML other than one of the five noted in the statement of the lemma. Thus, due to Lemma 7.5, there is at least one embedding of  $G$  in which every edge is adjacent to the exterior face. From the embeddings of  $G$  with this property, select one that maximizes the number of vertices on the exterior face, and let  $v$  denote a vertex that is not on this face. It must be that  $v$  has degree two, since otherwise  $G \geq_m K_4$  due to the way the embedding was chosen. Let  $u$  and  $w$  denote the vertices adjacent to  $v$ . The maximality of the embedding ensures that  $G$  contains three edge-disjoint paths of length two or more between  $u$  and  $w$ . Moreover, Lemma 7.6 implies that  $uw \in G$ .

Consider this embedding restricted to  $G' = G \setminus \{v\}$ . There are faces  $F_1$  and  $F_2$  in  $G'$  such that  $F_1 \cap F_2 = uw$ . Let  $M'$  denote a cost-three permutation for  $G'$  in which, due to Lemma 7.4, the overlap of the face spans for  $F_1$  and  $F_2$  is  $uw$ .

If  $uw$  contains no fill-in, then we construct a new matrix  $M$  from  $M'$  by adding row  $v$  and placing columns  $uv$  and  $vw$  to the immediate left of  $uw$ .

If  $uw$  contains a fill-in in some row  $x$ , then it follows that  $ux$  and  $wx$  must exist, contain the only 1s in row  $x$ , and lie immediately to each side of  $uw$ . In this case, we construct a new matrix  $M$  from  $M'$  by adding row  $v$  and placing columns  $uv$  and  $vw$  to the immediate left of the column to the immediate left of  $uw$ .

In either case,  $M$  is a cost-three permutation for  $G$ , contradicting the assumption that  $G$  is an obstruction for 3-GML.  $\square$

### 7.3. Additional properties of three-track obstructions

We shall henceforth consider only outerplane obstructions and outerplanar embeddings in which all vertices lie on the exterior face. Thus the intersection of two faces is at most a single edge.

**Lemma 7.8.** *If an obstruction for 3-GML contains two faces that are adjacent at and only connected through a single vertex, then at least one of these faces is a triangle with two vertices of degree two.*

**Proof.** Let  $G$  denote an obstruction, with faces  $F_1$  and  $F_2$  adjacent at and only connected through  $v$ . Assume neither  $F_1$  nor  $F_2$  is a triangle with two vertices of degree two.

Let  $C_1$  denote the (unique) connected component of  $G \setminus \{v\}$  that contains an edge of  $F_1 \setminus \{v\}$ . Let  $C_2$  denote  $(G \setminus \{v\}) \setminus C_1$ . Let  $u$  and  $w$  denote a pair of isolated vertices not in  $G$ . We define  $G_1 = (G \setminus C_2) \cup \{u, w\} \cup \{uv, vw, uw\}$  and  $G_2 = (G \setminus C_1) \cup \{u, w\} \cup \{uv, vw, uw\}$ .

Observe that both  $G_1$  and  $G_2$  are proper minors of  $G$  and both, therefore, have cost-three permutations. It is straightforward to show that  $G_1$  must possess an optimal permutation  $M_1$  with the three columns of  $\{u, v, w\}$  on the extreme right, else  $G$  properly contains an obstruction as described in Lemma 6.1. Similarly,  $G_2$  must possess an optimal permutation  $M_2$  with the three columns of  $\{u, v, w\}$  on the extreme left.

But this means that we can construct a cost-three permutation for  $G$  by placing  $M_2$  to the right of  $M_1$  and removing the (six) columns of  $\{u, v, w\}$ . This contradicts the fact that  $G$  is an obstruction, however, and so the assumption that neither  $F_1$  nor  $F_2$  is a triangle with two vertices of degree two cannot hold.  $\square$

Let  $v$  denote a vertex on a face of an outerplane graph  $G$ . If the connected component of  $G \setminus \{vw | w \text{ lies on a face}\}$  that contains  $v$  has at least one edge, then we term this component the *attachment* at  $v$ .

**Lemma 7.9.** *If an obstruction for 3-GML contains a face in which two or more vertices have attachments, then each attachment is a minor of  $S(K_{1,3})$ .*

**Proof.** Assume otherwise for obstruction  $G$ , in which vertices  $u$  and  $v$  of face  $F$  have attachments, with the attachment at  $v$ ,  $A(v)$ , not a minor of  $S(K_{1,3})$ .

No vertex of  $A(v)$  has degree greater than three unless  $A(v)$  contains a cycle (Lemmas 5.4 and 7.1). No degree-three vertex of  $A(v)$  is adjacent to both a vertex of degree two and a pendant vertex unless that vertex is  $v$  (Corollary 5.9). No degree-two vertex of  $A(v)$  is adjacent to two vertices of degree three (Lemma 7.6). It follows that either  $A(v)$  contains a cycle or  $S(K_{1,3}) <_m A(v)$ , and thus  $A(v)$  has cost three.

Let  $A^+ = A(v) \cup \{u, w\} \cup \{uv, vw, uw\}$ . Let  $A^- = A(v) \setminus \{v\}$ . It is straightforward to show that  $A^+$  possesses an optimal permutation  $M_1$  in which  $uvw^7$  is the rightmost column. Let  $G' = (G \setminus A^-) \cup \{x, y\} \cup \{xv, vy, xy\}$ . If  $A(v)$  contains a cycle, then  $G' <_m G$ , and thus  $G'$  has cost three. If  $A(v)$  is acyclic, then  $S(K_{1,3}) <_m A(v)$ , and thus (with the help of Lemma 5.16) again  $G'$  has cost three. Now it is straightforward to show that  $G'$  possesses an optimal permutation  $M_2$  in which  $vxy$  is the leftmost column. But this means we can construct a cost-three permutation for  $G$  by placing  $M_2$  to the right of  $M_1$  and removing  $uvw$  and  $vxy$ , a contradiction.  $\square$

**Lemma 7.10.** *If an obstruction for 3-GML contains two faces that are adjacent at and only connected through a single vertex, then there is an obstruction for 3-GML with one less face and with a vertex whose attachment is two or three pendant paths.*

**Proof.** Let  $G$  denote an obstruction with faces  $F_1$  and  $F_2$  adjacent at and only connected through  $v$ . Assume  $F_1$  is a triangle in which only  $v$  has degree three or more (Lemma 7.8). Let  $H$  denote the graph obtained from  $G$  by deleting  $F_1 \setminus \{v\}$  and identifying the degree-three vertex of (a disjoint copy of)  $S(K_{1,3})$  with  $v$ .  $H$  has cost four (Lemma 5.16). Let  $G'$  denote an obstruction contained in  $H$ . Observe that, in  $G'$ , the attachment at  $v$  contains more than one pendant path, else  $G' <_m G$ . Thus, due to Corollary 5.17, either  $G' = H$  or  $G' = H \setminus \{vx, xy\}$  where  $x$  and  $y$  are vertices on a pendant path incident on  $v$  and the lemma holds.  $\square$

**Corollary 7.11.** *If an obstruction for 3-GML contains two faces that are adjacent at and only connected through a single vertex  $v$ , then  $v$  has no attachment.*

We say that two disjoint faces are *separated* if the removal of some edge places the faces in different connected components.

**Lemma 7.12.** *If an obstruction for 3-GML contains a pair of separated faces, then the obstruction is one obtained from Lemma 6.1.*

<sup>7</sup> As justified by Lemma 5.13, we shall from now on represent a triangular face by a column with three 1s rather than three columns each with two 1s.

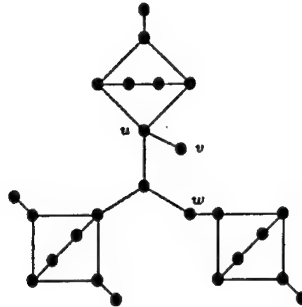


Fig. 5. A 4-GML obstruction.

**Proof.** Assume otherwise for obstruction  $G$  with separated faces  $F_1$  and  $F_2$ . Let  $uv$  denote an edge of  $G$  whose removal places  $F_1$  and  $F_2$  in distinct connected components  $C_1$  and  $C_2$ , respectively. Assume  $u \in C_1$  and  $v \in C_2$ .  $C_1$  must possess an optimal permutation  $M_1$  in which every column to the right of the span for  $u$  has cost two, else  $C_1 \setminus \{u\}$  contains two disjoint obstructions for 2-GML and the minimality of  $G$  ensures that it is obtained from Lemma 6.1. Similarly,  $C_2$  must possess an optimal permutation  $M_2$  in which every column to the left of the span for  $v$  has cost two. But now  $M_1, uv, M_2$  is a cost-three permutation for  $G$ , a contradiction.  $\square$

#### 7.4. Nonextendability of these results to four or more tracks

Unfortunately, the results of this section cannot be extended to values of  $k > 3$ . Consider, for example, the graph depicted in Fig. 5. We know from Lemma 6.1 that it is an obstruction for 4-GML.

Clearly, analogs of Lemmas 7.1 and 7.6 are ruled out by  $uv$  and  $w$ . Similarly, Lemma 6.2 quickly gives rise to obstructions for 4-GML that eliminate analogs for Lemmas 7.8 and 7.9. More complicated constructions [12] can be devised to rule out analogs for Lemmas 7.3, 7.4 and 7.12.

### 8. The complete three-track obstruction set

In this section, we shall complete the task of identifying all obstructions for 3-GML. Each is given a three-integer name, denoting its number of vertices, its number of interior faces and an index. For example, obstruction 8.2.3 is the third obstruction we list with eight vertices and two faces. For the reader's convenience, the entire set is displayed in an appendix to this paper.

#### 8.1. Obstructions from previous constructions

Lemma 6.1 provides twenty obstructions: ten are trees (22.0.1–10); six have one face (18.1.1–6); four have separated faces (10.3.1 and 14.2.1–3).

Lemma 6.2 provides forty-three more obstructions (6.4.1, 8.3.1, 9.4.1–2, 11.2.1–2, 11.3.1, 12.3.1, 13.2.1, 13.3.1–6, 15.1.1–4, 15.2.1–7, 16.2.1, 16.2.5–6, 17.1.1–3, 17.2.1, 17.2.3–4, 18.1.7, 18.1.9–10, 19.1.1–3, 20.1.1 and 21.1.1).

Lemma 6.4 provides one additional obstruction (15.1.5).

Therefore, including the five nonouterplanar obstructions identified in Section 7, sixty-nine obstructions for 3-GML are known up to this point.

## 8.2. Conventions for describing new obstructions

We know from [5, 12] and Lemma 7.12 that no more tree or separated-face obstructions are possible. Moreover, those with vertex-adjacent faces can be obtained indirectly with Lemma 7.10. Thus we now consider only outerplane graphs with either a single face or with two or more edge-adjacent faces. Without loss of generality, we assume the outerplane embedding induces a left-to-right ordering of the faces, so that we can employ a simple (decimal) integer *pattern* to denote its face structure. In such a pattern, the number of digits equals the number of faces, and the value of each digit equals the number of vertices in the corresponding face. (As we shall see later, this easy scheme suffices, because we need only consider candidate obstructions in which no interior face has more than six vertices.)

If a face contains four or more vertices, then we assume each vertex of the face has degree at least three (Lemmas 5.5, 5.6 and 7.6). If a vertex has an attachment, then we assume this attachment is either a pendant edge or one, two or three pendant paths (Lemma 7.9). If the attachment consists of three pendant paths, then a minimality-preserving replacement is possible thanks to Lemmas 5.16 and 7.6. We term this a *type 1 replacement*. If the attachment is a pendant edge, then a minimality-preserving replacement is possible thanks to Lemma 5.6. We term this a *type 2 replacement*. Fig. 6 illustrates these two replacements, which we shall use to identify obstructions that might otherwise be missed due to the assumptions just stated.

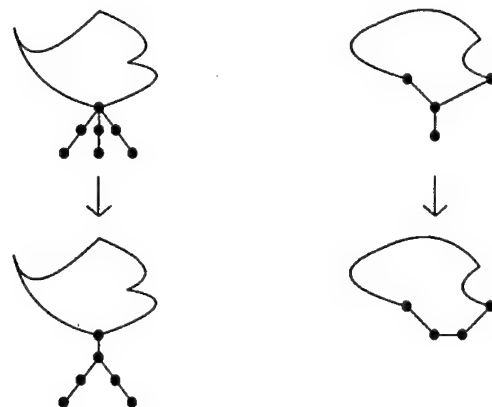


Fig. 6 Type 1 and 2 replacements.

We can thus use a succinct (character) *string* to denote a graph's attachment structure. We begin by visiting the vertices that lie on any internal face clockwise around the external face. If two or more (internal) faces are present, then we start with the vertex at the "top" of the edge shared by the leftmost two faces, otherwise we start at an arbitrary vertex. Letting  $v_i$  denote the  $i$ th vertex visited in this fashion, we represent the attachment at  $v_i$  with the  $i$ th character of the string. Such a character is either a 0 to denote that there is no attachment, the letter  $e$  to denote that it is a pendant edge, or an integer in the range  $[1, 3]$  to denote the number of pendant paths it contains.

New obstruction candidates are now uniquely (modulo rotations and reflections) describable in *pattern-string* form. For example, the graph denoted by 34-2e300 contains a triangle, edge adjacent to a square to its right. These two faces share the edge  $v_1v_4$ . The triangle's vertex set is  $\{v_1, v_4, v_5\}$ . The attachments at vertices  $v_1, v_2$ , and  $v_3$  are, respectively, two pendant paths, a pendant edge and three pendant paths.

In describing permutations of graphs, we adopt the convention that  $u_i$  denotes the other vertex of an edge pendant at  $v_i$ . It is also helpful to use a shorthand for (complete and partial) permutations of more complicated attachments. See Fig. 7. For example, if three pendant paths are incident on  $v$ , then we use  $A(v)$  in a permutation to indicate that the six edges of the attachment are to be placed in the order listed.

### 8.3. Obstructions with one face

*Triangular face.* If two vertices of the face have degree two, then it is straightforward to show that the graph can be obtained from Lemma 6.1. Otherwise, since the attachments at the vertices of the face are minors of  $S(K_{1,3})$ , the graph can be obtained from Lemma 6.2. Hereafter, we shall not consider any string that contains 333, 3321, 3312, 3213, 3123, 2133 or 1233, since the corresponding graph contains a minor whose pattern-string is 3-333 (known obstruction 21.1.1).

*Square face.* Pattern-string 4-2221 denotes new obstruction 18.1.8. Pattern-string 4-232e represents known obstruction 19.1.2. Any other graph with this pattern either contains one of these obstructions, or is a minor of a graph whose cost-three permutation resides in the following list.

4-3131  $A(v_1), C_1(v_2), v_1v_2, v_2v_3, v_1v_4, v_3v_4, C_2(v_4), A(v_3)$

4-323e  $A(v_1), B_1(v_2), v_1v_2, v_1v_4, u_4v_4, v_3v_4, v_2v_3, B_2(v_2), A(v_3)$

4-3311  $A(v_1), C_1(v_4), v_1v_4, v_1v_2, v_3v_4, v_2v_3, C_2(v_3), A(v_2)$

*Pentagonal face.* Pattern-strings 5-11111 and 5-22e1e correspond to known obstructions 15.1.5 and 17.1.1, respectively. Any new obstruction with a pentagonal face contains at least one, and at most two pendant edges. If a string has a single  $e$  and a single 1, then the corresponding graph contains obstruction 18.1.8 (4-2221). Any

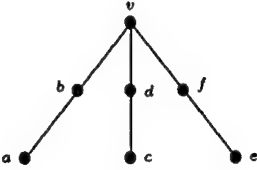
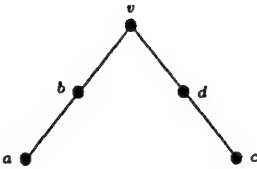

Attachment	Name	Notation	Permutation
	<i>A</i>	$A(v)$	<i>v</i> 0 1 * 1 1 0
			<i>a</i> 1 0 0 0 0 0
			<i>b</i> 1 1 0 0 0 0
			<i>c</i> 0 0 1 0 0 0
			<i>d</i> 0 0 1 1 0 0
			<i>e</i> 0 0 0 0 0 1
			<i>f</i> 0 0 0 0 1 1
	<i>B</i>	$B_1(v)$	<i>v</i> 0 1
			<i>a</i> 1 0
			<i>b</i> 1 1
		$B_2(v)$	<i>v</i> 1 0
			<i>c</i> 0 1
			<i>d</i> 1 1
	<i>C</i>	$C_1(v)$	<i>v</i> 0 1
			<i>a</i> 1 0
			<i>b</i> 1 1
		$C_2(v)$	<i>v</i> 1 0
			<i>a</i> 0 1
			<i>b</i> 1 1

Fig. 7. Shorthand used in permutations.

other candidate obstruction is a minor of a graph whose cost-three permutation resides in the following list.

5-3131e  $A(v_1), C_1(v_2), v_1v_2, v_1v_5, u_5v_5, v_4v_5, v_3v_4, v_2v_3, C_2(v_4), A(v_3)$

5-3113e  $A(v_1), C_1(v_2), v_1v_2, v_1v_5, u_5v_5, v_4v_5, v_3v_4, v_2v_3, C_2(v_3), A(v_4)$

5-1331e  $A(v_2), C_1(v_1), v_2v_3, v_1v_2, v_1v_5, u_5v_5, v_4v_5, v_3v_4, C_2(v_4), A(v_3)$

Hereafter, no string with five or more entries from  $\{1, 2, 3\}$  will be considered, because the corresponding graph contains known obstruction 15.1.5.

*Hexagonal face.* If three vertices of a graph with a hexagonal face have pendant edges incident on them, then the graph contains known obstruction 15.1.1 (6-1e1e1e). Thus we need only consider strings whose two *e* characters are in the third and sixth

positions. The graph with pattern-string 6-22e11e contains known obstruction 17.1.1 (5-22e1e). All other possibilities are minors of a graph whose cost-three permutation resides in the following list.

$$6-31e31e \quad A(v_1), C_1(v_2), v_1v_2, v_2v_3, u_3v_3, v_1v_6, u_6v_6, v_5v_6, v_3v_4, v_4v_5, C_2(v_5), A(v_4)$$

$$6-13e31e \quad A(v_2), C_1(v_1), v_1v_2, v_2v_3, u_3v_3, v_1v_6, u_6v_6, v_5v_6, v_3v_4, v_4v_5, C_2(v_5), A(v_4)$$

*Other faces.* Any graph that contains a face with seven or more vertices, each with an attachment, must contain either known obstruction 15.1.1 (6-1e1e1e) or known obstruction 15.1.5 (5-11111). An obstruction whose face contains seven or more vertices must therefore have adjacent vertices of degree two on the face, in which case the obstruction can be obtained from a type 2 replacement and has already been considered.

**Lemma 8.1.** *There are exactly 23 obstructions for 3-GML that contain only one face.*

In summary, only one new one-faced obstruction exists, bringing the total number of known obstructions up to 70.

#### 8.4. Obstructions with two faces

To identify obstructions with two vertex-adjacent faces, we apply the reverse of the replacement used in the proof of Lemma 7.10. Table 1 summarizes the two-faced obstructions thereby obtained. Other two-faced obstructions must contain edge-adjacent faces.

*Two triangles.* Pattern-string 33-0232 represents new obstruction 18.2.2, from which new obstruction 17.2.5 is obtained with a type 1 replacement. Pattern-string

Table 1  
Two-faced obstructions from Lemma 7.10

Starting one-faced obstruction	Resultant two-faced obstruction(s)
17.1.1	15.2.4
17.1.2	15.2.5, 15.2.6
17.1.3	15.2.7
18.1.8	15.2.2, 16.2.4 <sup>a</sup>
18.1.9	16.2.5
18.1.10	16.2.6
19.1.1	15.2.1
19.1.2	15.2.2, 17.2.3
19.1.3	15.2.3, 17.2.4
20.1.1	16.2.1
21.1.1	17.2.1

<sup>a</sup> New obstruction.

33-3230 denotes new obstruction 20.2.1, from which new obstructions 19.2.1 and 18.2.1 are obtained with type 1 replacements. The graph with pattern-string 33-2221 contains known obstruction 18.1.8 (4-2221). Pattern-string 33-2e22 represents new obstruction 17.2.6, from which new obstruction 17.2.7 is obtained with a type 2 replacement. All other possibilities are minors of a graph whose cost-three permutation resides in the following list.

33-0323  $A(v_2), B_1(v_3), v_1 v_2 v_3, v_1 v_3 v_4, B_2(v_3), A(v_4)$

33-1313  $A(v_2), C_1(v_1), v_1 v_2 v_3, v_1 v_3 v_4, C_2(v_3), A(v_4)$

33-3113  $A(v_1), C_1(v_2), v_1 v_2 v_3, v_1 v_3 v_4, C_2(v_3), A(v_4)$

33-3131  $A(v_1), C_1(v_2), v_1 v_2 v_3, v_1 v_3 v_4, C_2(v_4), A(v_3)$

33-3320  $A(v_2), B_1(v_3), v_1 v_2 v_3, v_1 v_3 v_4, B_2(v_3), A(v_1)$

*Triangle and square.* We assume the square is to the right of the triangle, so that both  $v_2$  and  $v_3$  must have attachments. If there is no  $e$  in the string, then there is at least one 0 in a position corresponding to a vertex of the triangle. Since known obstruction 13.2.1 has pattern-string 34-02200, we only consider graphs in which  $v_2$  or  $v_3$  has a pendant edge or a single pendant path as its attachment. A string with three 2s and a 1 corresponds to a graph that contains known obstruction 18.1.8 (4-2221). Pattern-string 34-21120 represents new obstruction 17.2.2. Pattern-string 34-2e102 denotes new obstruction 16.2.2, from which new obstruction 16.2.3 is obtained with a type 2 replacement. Pattern-string 34-1111e denotes new obstruction 14.2.7, from which new obstruction 14.2.8 is obtained with a type 2 replacement. Graphs with pattern-strings 34-2e230 and 34-2e232 contain known obstruction 19.1.2 (4-232e). The graph with pattern-string 34-1e22e contains known obstruction 17.1.1 (5-22e1e). All other possibilities are minors of a graph whose cost-three permutation resides in the following list.

34-0e323  $A(v_3), B_1(v_4), v_3 v_4, v_2 v_3, u_2 v_2, v_1 v_2, v_1 v_4 v_5, B_2(v_4), A(v_5)$

34-01313  $A(v_3), C_1(v_2), v_2 v_3, v_1 v_2, v_3 v_4, v_1 v_4 v_5, C_2(v_4), A(v_5)$

34-01331  $A(v_3), C_1(v_2), v_2 v_3, v_1 v_2, v_3 v_4, v_1 v_4 v_5, C_2(v_5), A(v_4)$

34-03113  $A(v_2), C_1(v_3), v_2 v_3, v_1 v_2, v_3 v_4, v_1 v_4 v_5, C_2(v_4), A(v_5)$

34-03131  $A(v_2), C_1(v_3), v_2 v_3, v_1 v_2, v_3 v_4, v_1 v_4 v_5, C_2(v_5), A(v_4)$

34-1e133  $A(v_4), C_1(v_3), v_3 v_4, v_2 v_3, u_2 v_2, v_1 v_2, v_1 v_4 v_5, C_2(v_1), A(v_5)$

34-1e313  $A(v_3), C_1(v_4), v_3 v_4, v_2 v_3, u_2 v_2, v_1 v_2, v_1 v_4 v_5, C_2(v_1), A(v_5)$

34-11330  $A(v_3), C_1(v_2), v_3 v_4, v_2 v_3, v_1 v_2, v_1 v_4 v_5, C_2(v_1), A(v_4)$

34-13130  $A(v_2), C_1(v_3), v_2 v_3, v_3 v_4, v_1 v_2, v_1 v_4 v_5, C_2(v_1), A(v_4)$

34-3e131  $A(v_4), C_1(v_3), v_3 v_4, v_2 v_3, u_2 v_2, v_1 v_2, v_1 v_4 v_5, C_2(v_5), A(v_1)$

34-3e311  $A(v_3), C_1(v_4), v_3 v_4, v_2 v_3, u_2 v_2, v_1 v_2, v_1 v_4 v_5, C_2(v_5), A(v_1)$

34-3e320  $A(v_3), B_1(v_4), v_3 v_4, v_2 v_3, u_2 v_2, v_1 v_2, v_1 v_4 v_5, B_2(v_4), A(v_1)$

*Two squares.* Pattern-string 44-1e101e denotes new obstruction 14.2.4, from which new obstructions 14.2.5 and 14.2.6 are obtained with type 2 replacements. Graphs with pattern-strings 44-2e10e2 and 44-0e22e1 contain known obstruction 17.1.1 (5-22e1e). The graph with pattern-string 44-0e2320 contains known obstruction 19.1.2 (4-232e). If a string contains no  $e$ , then its first and fourth characters must both be 0 (Lemma 7.1 and avoidance of known obstruction 15.1.5 (5-11111)). Known obstruction 13.2.1 (34-02200) is a minor of any graph with pattern 44 in which both  $v_2$  and  $v_3$  (or both  $v_5$  and  $v_6$ ) have two or more pendant paths as attachments. All other possibilities are minors of a graph whose cost-three permutation resides in the following list.

- 44-0e1313  $A(v_4), C_1(v_3), v_3v_4, v_2v_3, u_2v_2, v_1v_2, v_1v_4, v_4v_5, v_1v_6, v_5v_6, C_2(v_5), A(v_6)$
- 44-0e1331  $A(v_4), C_1(v_3), v_3v_4, v_2v_3, u_2v_2, v_1v_2, v_1v_4, v_4v_5, v_1v_6, v_5v_6, C_2(v_6), A(v_5)$
- 44-0e3113  $A(v_3), C_1(v_4), v_3v_4, v_2v_3, u_2v_2, v_1v_2, v_1v_4, v_4v_5, v_1v_6, v_5v_6, C_2(v_5), A(v_6)$
- 44-0e3131  $A(v_3), C_1(v_4), v_3v_4, v_2v_3, u_2v_2, v_1v_2, v_1v_4, v_4v_5, v_1v_6, v_5v_6, C_2(v_6), A(v_5)$
- 44-0e323e  $A(v_3), B_1(v_4), v_3v_4, v_2v_3, u_2v_2, v_1v_2, v_1v_4, v_1v_6, u_6v_6, v_5v_6, v_4v_5, B_2(v_4), A(v_5)$
- 44-0e331e  $A(v_3), v_3v_4, v_2v_3, u_2v_2, v_1v_2, v_1v_4, v_1v_6, u_6v_6, v_5v_6, v_4v_5, C_2(v_5), A(v_4)$
- 44-031031  $A(v_2), C_1(v_3), v_2v_3, v_1v_2, v_3v_4, v_1v_4, v_1v_6, v_4v_5, v_5v_6, C_2(v_6), A(v_5)$
- 44-031013  $A(v_2), C_1(v_3), v_2v_3, v_1v_2, v_3v_4, v_1v_4, v_1v_6, v_4v_5, v_5v_6, C_2(v_5), A(v_6)$
- 44-1e31e3  $A(v_3), C_1(v_4), v_3v_4, v_2v_3, u_2v_2, v_1v_2, v_1v_4, v_4v_5, u_5v_5, v_5v_6, v_1v_6, C_2(v_1), A(v_6)$
- 44-3e13e1  $A(v_4), C_1(v_3), v_3v_4, v_2v_3, u_2v_2, v_1v_2, v_1v_4, v_4v_5, u_5v_5, v_5v_6, v_1v_6, C_2(v_6), A(v_1)$
- 44-3e31e1  $A(v_3), C_1(v_4), v_3v_4, v_2v_3, u_2v_2, v_1v_2, v_1v_4, v_4v_5, u_5v_5, v_5v_6, v_1v_6, C_2(v_6), A(v_1)$

*Other patterns.* The next result ensures that all two-faced obstructions with other patterns are already known (either by Lemma 6.2 or by type 2 replacements).

**Lemma 8.2.** *Obstruction 11.2.1 is the only two-faced outerplane obstruction for 3-GML with edge-adjacent faces in which one face has five or more vertices each with degree at least three.*

**Proof.** Assume otherwise for some obstruction  $G$  with faces  $F_1$  and  $F_2$ , where  $F_1 \cap F_2 = v_1v_m$ ,  $m \geq 5$ , and vertices  $v_2, v_3, \dots, v_{m-1}$  of  $F_2$  each has an attachment. Since  $G$  does not by assumption contain obstruction 11.2.1 (35-01e100), the attachment at  $v_2$  or  $v_4$  must be a pendant edge, and the attachment at  $v_3$  must be one or more pendant paths.

Suppose the attachment at  $v_2$  is the pendant edge  $u_2v_2$ . Let  $G' = G \setminus \{u_2v_2\}$ . Thanks to Lemma 7.4,  $G'$  possesses a cost-three permutation  $M'$  in which the overlap of the face spans for  $F_1$  and  $F_2$  is column  $v_1v_m$ , the leftmost column of  $F_2$ . If the attachment at  $v_4$  contains one or more pendant paths, then  $v_1v_2$  must be the rightmost column in

the span for  $v_1$ . But this means that a cost-three permutation for  $G$  can be constructed from  $M'$ , a contradiction. Thus the attachment at  $v_4$  is a pendant edge. It follows that  $F_2$  must be a pentagon (else  $v_5$  has an attachment with one or more pendant paths and  $G$  properly contains obstruction 11.2.1). Additionally, both  $v_1$  and  $v_5$  must have attachments, since otherwise  $M'$  can again be modified to produce a cost-three permutation for  $G$ . It is now clear that  $F_1$  must be a triangle with vertex set  $\{v_1, v_5, v_6\}$ , and that  $v_6$  must have degree two, else  $G$  properly contains obstruction 15.1.1 (6-1e1e1e). Also, the attachment at  $v_1$  or  $v_5$  must be a single pendant path, else  $G$  contains obstruction 17.1.1 (5-2e1e2). But this means that  $G$  is a minor of the graph with pattern-string 35-3e3e10, which has cost-three permutation  $A(v_1), C_1(v_5), v_1v_5v_6, v_1v_2, u_2v_2, v_4v_5, u_4v_4, v_3v_4, v_2v_3, A(v_3)$ , again a contradiction.

Suppose the attachment at  $v_2$  is one or more pendant paths. The attachment at  $v_4$  must be a pendant edge, from which it again follows that  $F_2$  must be a pentagon, reducing this by symmetry to the previous case.  $\square$

**Lemma 8.3.** *There are 39 obstructions for 3-GML that have exactly two faces.*

In summary, sixteen new two-faced obstructions exist, bringing the total number of known obstructions up to 86.

### 8.5. Obstructions with three faces

To identify obstructions with three faces some of which are adjacent at and only connected through a single vertex, we again apply the reverse of the replacement used in the proof of Lemma 7.10. Table 2 summarizes the three-faced obstructions thereby obtained.

In any additional three-faced obstruction, each face must be edge adjacent to at least one other. Furthermore, the three faces cannot be mutually edge adjacent, else the graph contains  $K_4$ .

**Lemma 8.4.** *No outerplane obstruction for 3-GML contains faces,  $F_1$ ,  $F_2$ , and  $F_3$ , where  $F_1 \cap F_3 = \emptyset$ , such that both  $F_1$  and  $F_3$  are edge adjacent to  $F_2$ .*

**Proof.** Assume otherwise for some obstruction  $G$  with faces  $F_1, F_2$ , and  $F_3$  for which  $F_1 \cap F_2 = v_1v_r$  and  $F_2 \cap F_3 = v_iv_j$ , where  $1 < i < j < r$ .

Suppose  $F_2$  is a square with vertex set  $\{v_1, v_2, v_{r-1}, v_r\}$ . Let  $G' = G \setminus \{v_1v_r\}$ , and let  $F'_2$  denote the (enlarged) face that results from the removal of  $v_1v_r$  from  $F_2$ .  $G'$  possesses a cost-three permutation  $M'$  in which the overlap of the spans for  $F'_2$  and  $F_3$  is column  $v_2v_{r-1}$ , the leftmost column of  $F_3$ . If both  $v_1$  and  $v_r$  have attachments, then their spans must include the leftmost column of  $F'_2$ , and  $v_1v_2$  can be placed to the immediate left of the span for  $F'_2$  to obtain a cost-three permutation for  $G$ , a contradiction. Thus  $v_1$  or  $v_r$  (and analogously  $v_2$  or  $v_{r-1}$ ) has no attachment. If neither  $v_2$  nor  $v_r$  has an attachment, then  $v_1v_2$  can be moved to the immediate left of  $v_2v_{r-1}$  and

Table 2  
Three-faced obstructions from Lemma 7.10

Starting two-faced obstruction	Resultant three-faced obstruction(s)
13.2.1	11.3.1
15.2.2	13.3.5
15.2.3	13.3.6
15.2.4	13.3.2
15.2.5	13.3.3
15.2.6	13.3.3
15.2.7	13.3.4
16.2.1	12.3.1
16.2.2	14.3.3 <sup>a</sup> , 14.3.5 <sup>a</sup>
16.2.3	14.3.4 <sup>a</sup> , 14.3.6 <sup>a</sup>
16.2.4	13.3.1, 14.3.2 <sup>a</sup>
16.2.5	12.3.1
16.2.6	12.3.1
17.2.1	13.3.1
17.2.2	15.3.1 <sup>a</sup>
17.2.3	13.3.1, 13.3.5
17.2.4	13.3.1, 13.3.6
17.2.5	15.2.1
17.2.6	15.2.2, 15.3.3 <sup>a</sup>
17.2.7	15.2.3, 15.3.4 <sup>a</sup>
18.2.1	15.2.1
18.2.2	14.3.1 <sup>a</sup> , 16.2.1
19.2.1	15.3.2 <sup>a</sup> , 16.2.1
20.2.1	16.3.1 <sup>a</sup> , 17.2.1

<sup>a</sup> New obstruction.

$v_{r-1}v_r$  can be moved to the immediate left of  $v_1v_2$ , making it easy to construct a cost-three permutation for  $G$ , a contradiction. Thus  $v_2$  or  $v_r$  (and analogously  $v_1$  or  $v_{r-1}$ ) has an attachment. So, without loss of generality, assume both  $v_1$  and  $v_2$  have attachments. Let  $G''$  denote the graph obtained from  $G$  by contracting edge  $v_{r-1}v_r$  to  $v_r$ , and let  $F''$  denote the triangle with vertex set  $\{v_1, v_2, v_r\}$ .  $G''$  possesses a cost-three permutation  $M''$  in which  $v_1v_2$  lies between  $v_1v_r$ , the rightmost column of  $F_1$ , and  $v_2v_r$ , the leftmost column of  $F_3$ .  $M''$  can now be modified by adding row  $v_{r-1}$ , replacing  $v_2v_r$  by  $v_{r-1}v_r$  and  $v_2v_{r-1}$ , and, in every column to the right of  $v_2v_{r-1}$ , interchanging the contents of rows  $v_r$  and  $v_{r-1}$ , thereby producing a cost-three permutation for  $G$ , a contradiction.

$F_2$  must therefore have five or more vertices. Without loss of generality, assume  $v_2$  does not lie on  $F_3$  and has degree three or more. The attachment at  $v_2$  must be the pendant edge  $u_2v_2$  and  $v_3$  must lie on  $F_3$ , else  $G$  contains obstruction 8.3.1 (343-010000). But now it is a simple matter to modify a cost-three permutation for  $G \setminus \{u_2v_2\}$  to obtain a cost-three permutation for  $G$ , again a contradiction.  $\square$

*Three triangles.* Since known obstruction 13.2.1 has pattern-string 34-02200, and since removal of  $v_1 v_4$  (or  $v_2 v_4$ ) leaves an edge-adjacent triangle and square, we do not consider any graph in which both  $v_2$  and  $v_3$  (or both  $v_1$  and  $v_5$ ) have two or more pendant paths as attachments. Any graph with attachments at all five vertices contains new obstruction 13.3.7 denoted by pattern-string 333-11e1e, from which new obstructions 13.3.8 and 13.3.9 are obtained with type 2 replacements. Pattern-string 333-21e20 denotes new obstruction 16.3.2, from which new obstruction 16.3.3 is obtained with a type 2 replacement. Pattern-string 333-22030 denotes new obstruction 19.3.1, from which new obstruction 18.3.1 is obtained with a type 1 replacement. Graphs with pattern-strings 333-00232 and 333-02032 contain known obstruction 18.2.2 (33-0232). The graph with pattern-string 333-12021 contains known obstruction 17.2.2 (34-21120). All other possibilities are minors of a graph whose cost-three permutation resides in the following list.

- 333-00323  $A(v_3), B_1(v_4), v_2 v_3 v_4, v_1 v_2 v_4, v_1 v_4 v_5, B_2(v_4), A(v_5)$
- 333-01313  $A(v_3), C_1(v_2), v_2 v_3 v_4, v_1 v_2 v_4, v_1 v_4 v_5, C_2(v_4), A(v_5)$
- 333-01331  $A(v_3), C_1(v_2), v_2 v_3 v_4, v_1 v_2 v_4, v_1 v_4 v_5, C_2(v_5), A(v_4)$
- 333-03023  $A(v_2), B_1(v_4), v_2 v_3 v_4, v_1 v_2 v_4, v_1 v_4 v_5, B_2(v_4), A(v_5)$
- 333-03113  $A(v_2), C_1(v_3), v_2 v_3 v_4, v_1 v_2 v_4, v_1 v_4 v_5, C_2(v_4), A(v_5)$
- 333-03131  $A(v_2), C_1(v_3), v_2 v_3 v_4, v_1 v_2 v_4, v_1 v_4 v_5, C_2(v_5), A(v_4)$
- 333-11033  $A(v_4), C_1(v_2), v_2 v_3 v_4, v_1 v_2 v_4, v_1 v_4 v_5, C_2(v_1), A(v_5)$
- 333-11303  $A(v_3), C_1(v_2), v_2 v_3 v_4, v_1 v_2 v_4, v_1 v_4 v_5, C_2(v_1), A(v_5)$
- 333-13013  $A(v_2), C_1(v_4), v_2 v_3 v_4, v_1 v_2 v_4, v_1 v_4 v_5, C_2(v_1), A(v_5)$
- 333-13103  $A(v_2), C_1(v_3), v_2 v_3 v_4, v_1 v_2 v_4, v_1 v_4 v_5, C_2(v_1), A(v_5)$
- 333-31031  $A(v_4), C_1(v_2), v_2 v_3 v_4, v_1 v_2 v_4, v_1 v_4 v_5, C_2(v_5), A(v_1)$
- 333-33011  $A(v_2), C_1(v_4), v_2 v_3 v_4, v_1 v_2 v_4, v_1 v_4 v_5, C_2(v_5), A(v_1)$
- 333-33020  $A(v_2), B_1(v_4), v_2 v_3 v_4, v_1 v_2 v_4, v_1 v_4 v_5, B_2(v_4), A(v_1)$
- 333-33101  $A(v_2), C_1(v_3), v_2 v_3 v_4, v_1 v_2 v_4, v_1 v_4 v_5, C_2(v_5), A(v_1)$

*Other patterns.* The next result ensures that all three-faced obstructions with other patterns are already known (either by Lemma 6.2 or by type 2 replacements).

**Lemma 8.5.** *Obstruction 8.3.1 is the only three-faced outerplane obstruction for 3-GML in which each face is edge adjacent to at least one other and one face has four or more vertices each with degree at least three.*

**Proof.** Assume otherwise for some obstruction  $G$  with faces  $F_1, F_2$ , and  $F_3$  such that both  $F_1$  and  $F_3$  are edge adjacent to  $F_2$ . Thanks to lemma 8.4, we may assume  $F_1 \cap F_2 = v_1 v_r$ , and  $F_2 \cap F_3 = v_i v_r$ .

Suppose  $F_2$  is not a triangle. To avoid obstruction 8.3.1 (343-010000),  $F_2$  must be a square with vertex set  $\{v_1, v_2, v_3, v_r\}$ , and  $v_2$  must be adjacent to pendant vertex  $u_2$ . Let  $G' = G \setminus \{u_2 v_2\}$ .  $G'$  possesses a cost-three permutation  $M'$  in which  $v_1 v_2$  and  $v_2 v_3$  lie between  $v_1 v_r$ , the rightmost column of  $F_1$ , and  $v_3 v_r$ , the leftmost column of  $F_3$ . It is straightforward to verify that  $v_1 v_2$  contains the rightmost 1 in row  $v_1$ , that  $v_2 v_3$  is to the immediate right of  $v_1 v_2$ , and that  $u_2 v_2$  can be inserted in  $M'$  to produce a cost-three permutation for  $G$ , a contradiction.

Thus  $F_2$  must be a triangle. Without loss of generality, assume  $F_3$  has at least four vertices each with degree at least three. If  $v_2$  has an attachment, then to avoid obstruction 11.2.1 (35-01e100) it follows that  $F_3$  must be a square with vertex set  $\{v_2, v_3, v_4, v_5\}$ , the attachment at  $v_4$  is the pendant edge  $u_4 v_4$ , and the attachment at  $v_3$  contains at least one pendant path. Let  $G'' = G \setminus \{u_4 v_4\}$  and let  $M''$  denote a cost-three permutation for  $G''$  in which the span for  $F_3$  is to the right of column  $v_1 v_2 v_5$ . Since  $v_4 v_5$  must be the rightmost column in the span for  $v_5$ , it is straightforward to construct a cost-three permutation for  $G$ , a contradiction. Thus  $v_2$  can have no attachment. Let  $G'''$  denote the graph obtained from  $G$  by contracting edge  $v_2 v_3$  to  $v_2$ , and let  $F_3'''$  denote the (shrunk) face that results from this contraction in  $F_3$ . Using a cost-three permutation for  $G'''$  in which the span for  $F_3'''$  is to the right of column  $v_1 v_2 v_r$ , it is again straightforward to construct a cost-three permutation for  $G$ , a contradiction.  $\square$

**Lemma 8.6.** *There are 29 obstructions for 3-GML that have exactly three faces.*

In summary, eighteen new three-faced obstructions exist, bringing the total number of known obstructions up to 104.

#### 8.6. Obstructions with four faces

To identify obstructions with four faces some of which are adjacent at and only connected through a single vertex, we again apply the reverse of the replacement used in the proof of Lemma 7.10. Table 3 summarizes the four-faced obstructions thereby obtained.

In any additional four-faced obstruction, each face must be edge adjacent to at least one other. One face cannot be edge adjacent to the other three, else the graph contains known obstruction 6.4.1. Furthermore, to avoid  $K_4$ , at least two faces must be edge adjacent to exactly one other face. Our next result ensures that all four-faced obstructions are already known.

A *chain* in an outerplane graph is a sequence of faces  $F_1, F_2, \dots, F_h$  such that  $F_i$  and  $F_j$  intersect at a single edge if  $|i - j| = 1$ , and are either disjoint or intersect at a single vertex otherwise. The length of a chain is the number of faces it contains. Fig. 8 illustrates four different four-faced chains.

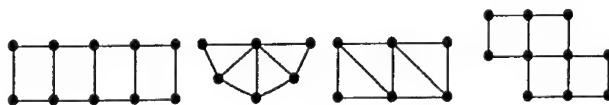


Fig. 8. Sample four-faced chains.

Table 3  
Four-faced obstructions from Lemma 7.10

Starting three-faced obstruction	Resultant four-faced obstruction(s)
11.3.1	9.4.2
13.3.1	9.4.1
13.3.5	9.4.1
13.3.6	9.4.1
14.3.1	12.3.1
14.3.2	9.4.1
14.3.3	12.3.1
14.3.4	12.3.1
14.3.5	12.3.1
14.3.6	12.3.1
15.3.1	12.4.1 <sup>a</sup>
15.3.2	12.3.1
15.3.3	12.4.1 <sup>a</sup> , 13.3.1
15.3.4	12.4.1 <sup>a</sup> , 13.3.1
16.3.1	12.4.1 <sup>a</sup> , 13.3.1
16.3.2	14.4.1 <sup>a</sup> , 14.4.3 <sup>a</sup>
16.3.3	14.4.2 <sup>a</sup> , 14.4.4 <sup>a</sup>
18.3.1	15.3.2
19.3.1	15.4.1 <sup>a</sup> , 16.3.1

<sup>a</sup>New obstruction.

**Lemma 8.7.** *No obstruction for 3-GML contains a chain whose length exceeds three.*

**Proof.** Assume otherwise for some obstruction  $G$  with chain  $F_1, F_2, \dots, F_h$  where  $h \geq 4$  and  $F_i \cap F_{i+1} = v_i w_i$  for  $1 \leq i < h$ .

Thanks to Lemma 8.4, we assume without loss of generality that  $w_1 = w_2$ . To avoid obstruction 8.3.1,  $G$  must contain either  $v_1 v_2$  or a degree-three vertex  $x$  adjacent to  $v_1, v_2$  and pendant vertex  $y$ .

Let  $G' = G \setminus \{v_2 w_2\}$ , and let  $F'_2$  denote the (enlarged) face that results from the removal of  $v_2 w_2$  from  $F_2$ .  $G'$  possesses a cost-three permutation in which the overlap of the spans for  $F_1$  and  $F'_2$  is  $v_1 w_1$ , the leftmost column of  $F'_2$ . Since any attachment at  $v_1$  must lie to the left of the span for  $F_1$ , since the span for  $F_4$  must be to the right of  $v_1 w_1$ , and since outerplanarity ensures  $v_1 \notin F_4$ , column  $v_1 v_2$  (or column  $v_1 x$ ) must contain the rightmost 1 in row  $v_1$ . Thus, with no increase in cost, column  $v_1 v_2$  (or the set of columns  $a \ v_1 x, xy, xv_2$ ) may be moved to the immediate right of  $v_1 w_1$ , from which it is straightforward to construct a cost-three permutation for  $G$ , a contradiction.  $\square$

**Lemma 8.8.** *There are nine obstructions for 3-GML that have exactly four faces.*

In summary, six new four-faced obstructions exist, bringing the total number of known obstructions up to 110. We shall now show that there are no more.

Table 4  
A review of the 3-GML  
obstruction set

Number of faces	Number of obstructions
none	10
one	23
two	39
three	29
four	9
five or more	0

### 8.7. Obstructions with five or more faces

**Lemma 8.9.** *No obstruction for 3-GML contains five or more faces.*

**Proof.** The reverse of the replacement used in the proof of Lemma 7.10 generates only known obstructions 9.4.1 and 12.4.1. Thus there can be no obstruction with five or more faces some of which are adjacent at and only connected through a single vertex. Thanks to Lemmas 7.8 and 8.7, no obstruction can contain either separated faces or a chain whose length exceeds three.  $\square$

## 9. Main result

All elements of the 3-GML obstruction set are now known. The structure of this set is reviewed in Table 4.

**Theorem 9.1.** *There are exactly 110 obstructions for 3-GML, namely, those identified in preceding results and depicted in the appendix.*

## 10. Conclusions

Gate matrix layout is a well-known but notoriously difficult problem. Each of its fixed-parameter variants, however, possesses a finite-basis characterization that provides a polynomial-time recognition algorithm. In this paper, we have isolated the basis for parameter value three. In order to accomplish this, we have also derived a number of more general results to bound and identify basis elements for any parameter value.

We conjecture that the trees are the largest elements in each basis. A proof of this, if it is indeed true, would be particularly interesting, because it would automatically mean that every basis is computable. (Exhaustive computation could, at least in

principle, be applied until the trees were reached, after which it would be pointless to look further.)

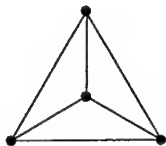
Lemma 6.1 makes it easy to see that basis size grows monotonically. This and the fact that the basis for parameter value four contains at least 122 million elements [12] suggest that no other bases for this problem are likely to be isolated in the foreseeable future.

## References

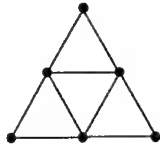
- [1] J.A. Bondy and U.S.R. Murty, *Graph Theory with Applications* (North-Holland, New York, 1976).
- [2] R.L. Bryant, M.R. Fellows, N.G. Kinnersley and M.A. Langston, On finding obstruction sets and polynomial-time algorithms for gate matrix layout, in: *Proceedings of the 25th Allerton Conference on Communication, Control, and Computing* (1987) 397–398.
- [3] D.J. Brown, M.R. Fellows and M.A. Langston, Polynomial-time self-reducibility: theoretical motivations and practical results, *Internat. J. Comput. Math.* 31 (1989) 1–9.
- [4] N. Deo, M.S. Krishnamoorthy and M.A. Langston, Exact and approximate solutions for the gate matrix layout problem, *IEEE Trans. Computer-Aided Design* 6 (1987) 79–84.
- [5] J.A. Ellis, I.H. Sudborough and J.S. Turner, The vertex separation and search number of a graph, *Inform. and Comput.*, to appear.
- [6] M.R. Fellows and M.A. Langston, Nonconstructive advances in polynomial-time complexity, *Inform. Process. Lett.* 26 (1987) 157–162.
- [7] M.R. Fellows and M.A. Langston, Nonconstructive tools for providing polynomial-time decidability, *J. ACM* 35 (1988) 727–739.
- [8] M.R. Fellows and M.A. Langston, On well-partial-order theory and its application to combinatorial problems of VLSI design, *SIAM J. Discrete Math.* 5 (1992) 117–126.
- [9] M.R. Fellows and M.A. Langston, Fast search algorithms for layout permutation problems, *Internat. J. Comput. Aided VLSI Design* 3 (1991) 325–342.
- [10] M.R. Fellows and M.A. Langston, On search, decision and the efficiency of polynomial-time algorithms, in: *Proceedings of the 21st ACM Symposium on the Theory of Computing* (1989) 501–512.
- [11] F. Harary, *Graph Theory* (Addison-Wesley, Reading, MA, 1969).
- [12] N.G. Kinnersley, Obstruction set isolation for layout permutation problems, Ph.D. Thesis, Washington State University (1989).
- [13] C. Kuratowski, Sur le probleme des courbes gauches en topologie, *Fund. Math.* 15 (1930) 271–283.
- [14] N. Robertson and P.D. Seymour, Graph minors I. Excluding a forest, *J. Combin. Theory Ser. B* 35 (1983) 39–61.
- [15] N. Robertson and P.D. Seymour, Graph minors V. Excluding a planar graph, *J. Combin. Theory Ser. B* 41 (1986) 92–114.
- [16] N. Robertson and P.D. Seymour, Graph minors XIII. The disjoint paths problems, to appear.
- [17] N. Robertson and P.D. Seymour, Graph minors XVI. Wagner's conjecture, to appear.
- [18] P.D. Seymour, private communication.

## Appendix

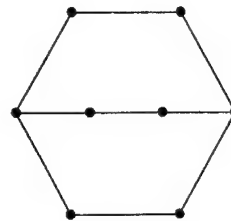
### The 3-GML Obstruction Set



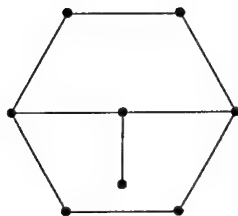
4.3.1



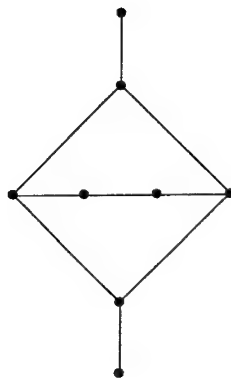
6.4.1



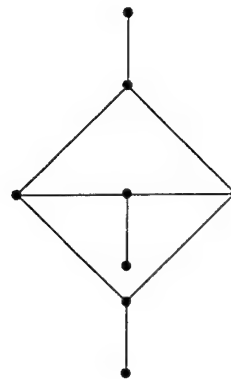
8.2.1



8.2.2



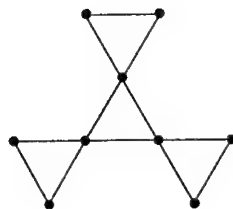
8.2.3



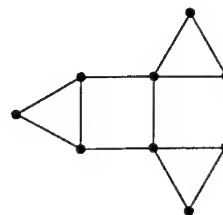
8.2.4



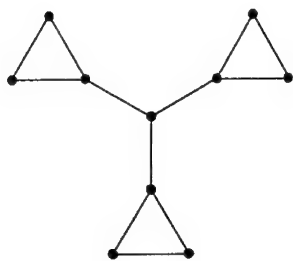
8.3.1



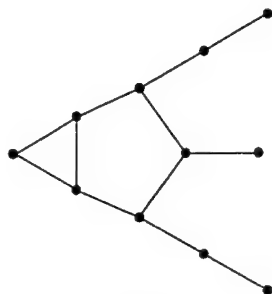
9.4.1



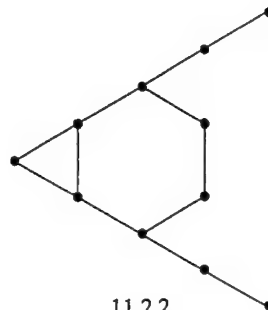
9.4.2



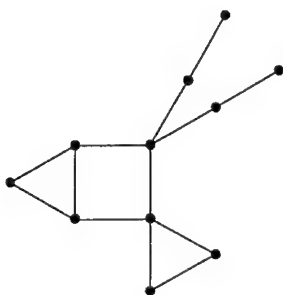
10.3.1



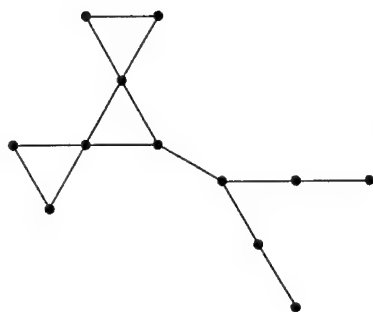
11.2.1



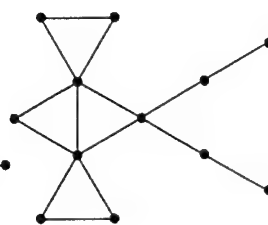
11.2.2



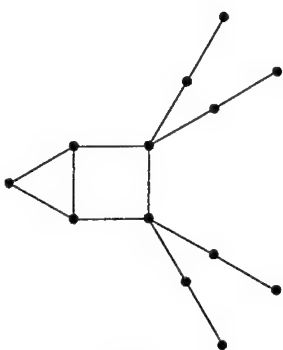
11.3.1



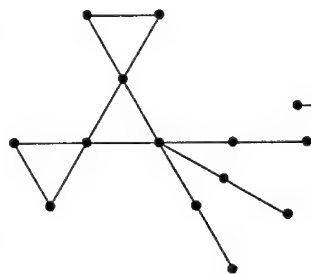
12.3.1



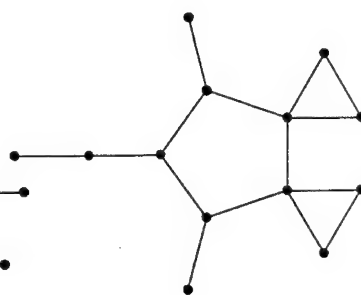
12.4.1



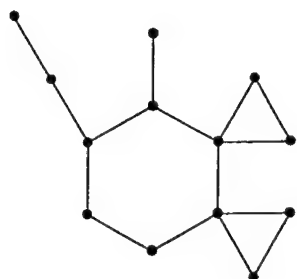
13.2.1



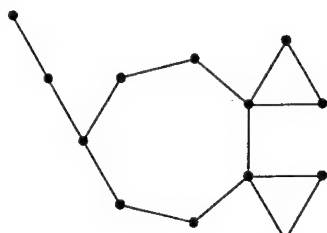
13.3.1



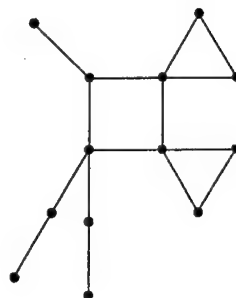
13.3.2



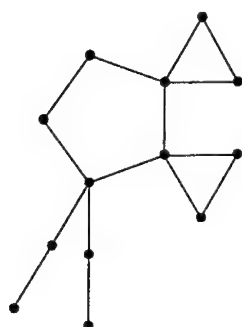
13.3.3



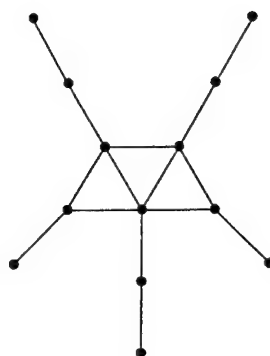
13.3.4



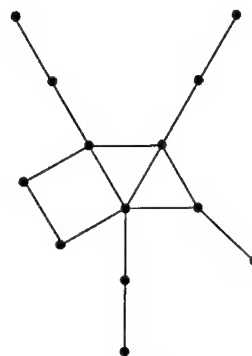
13.3.5



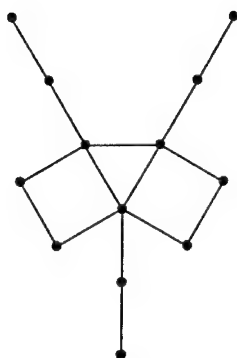
13.3.6



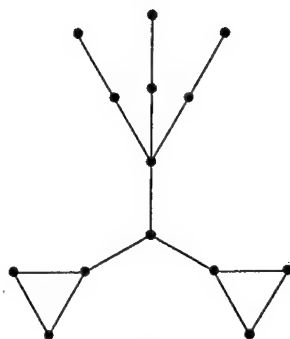
13.3.7



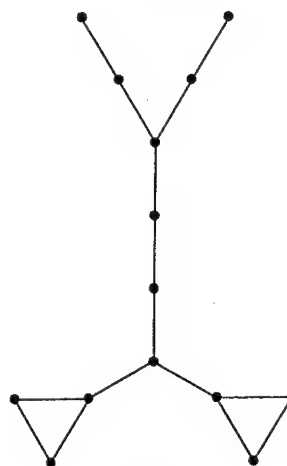
13.3.8



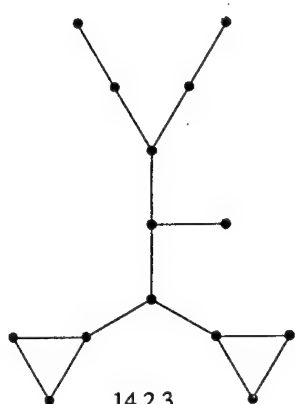
13.3.9



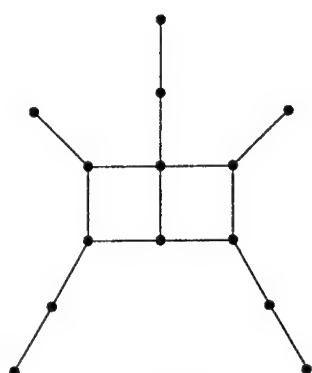
14.2.1



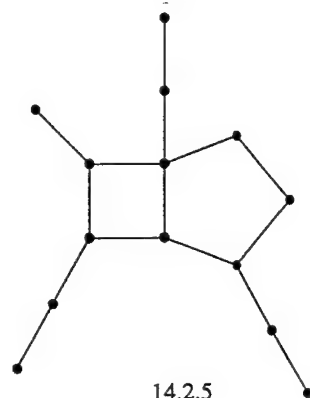
14.2.2



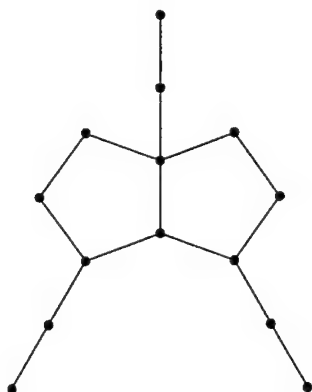
14.2.3



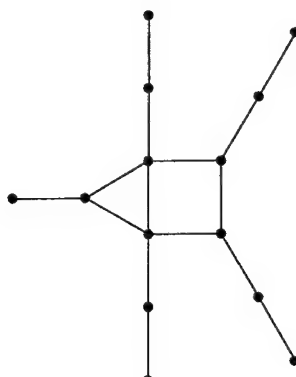
14.2.4



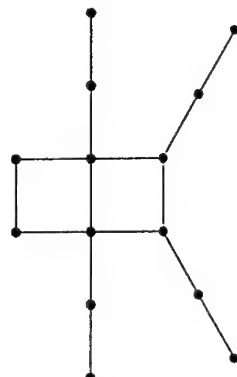
14.2.5



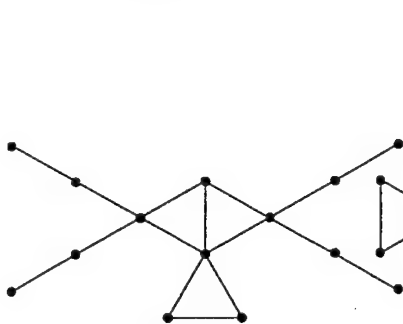
14.2.6



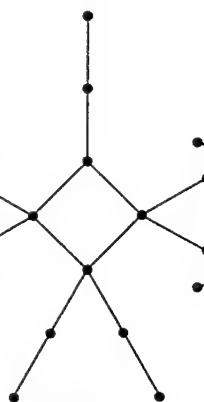
14.2.7



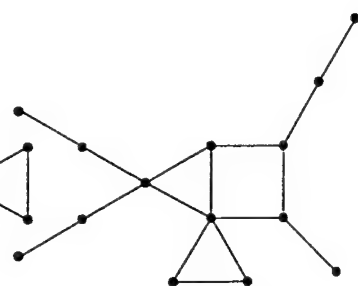
14.2.8



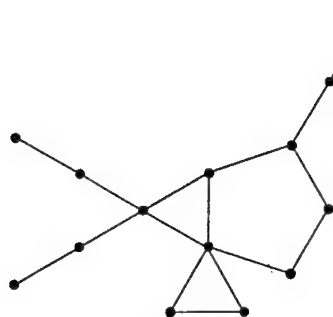
14.3.1



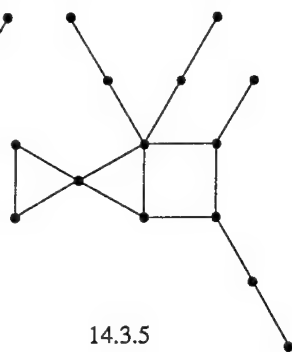
14.3.2



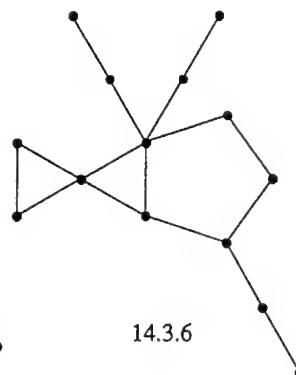
14.3.3



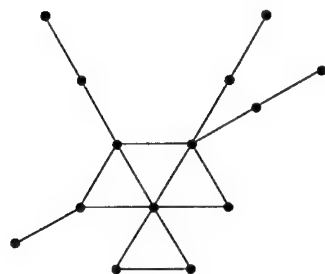
14.3.4



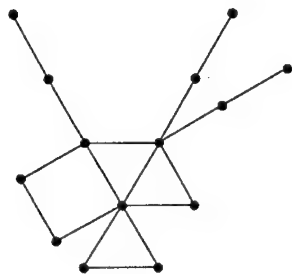
14.3.5



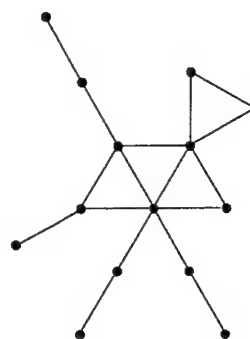
14.3.6



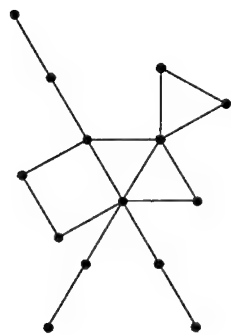
14.4.1



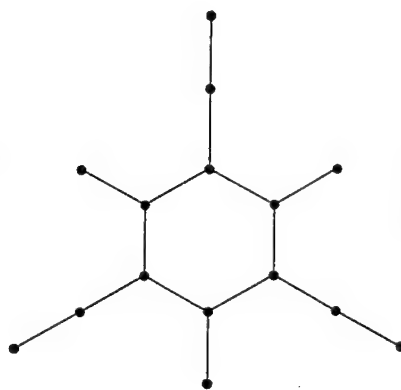
14.4.2



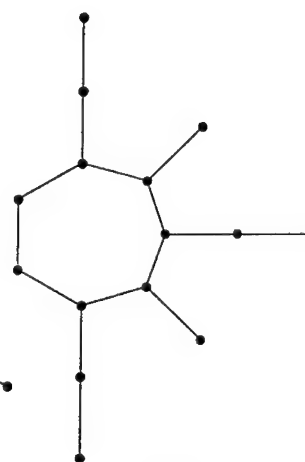
14.4.3



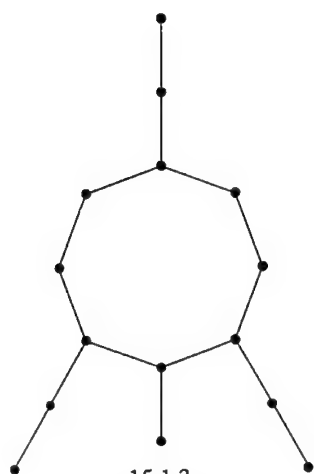
14.4.4



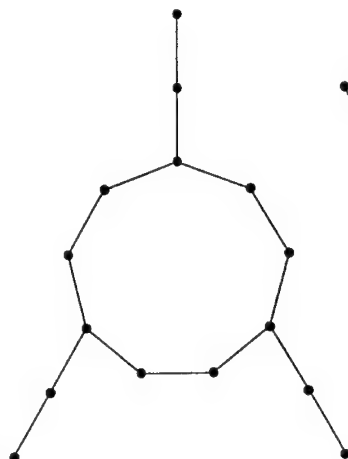
15.1.1



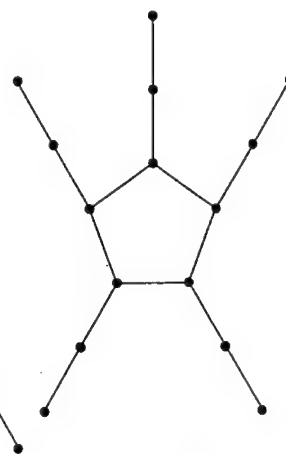
15.1.2



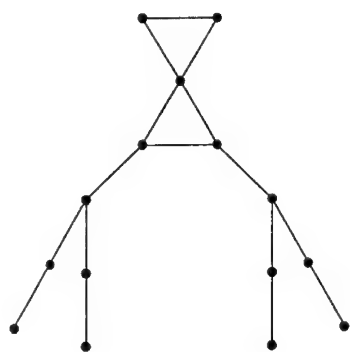
15.1.3



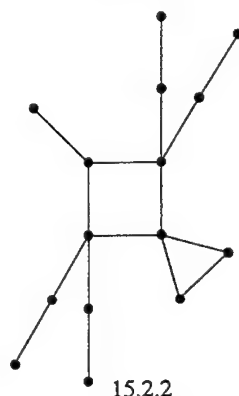
15.1.4



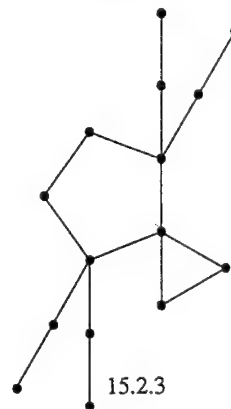
15.1.5



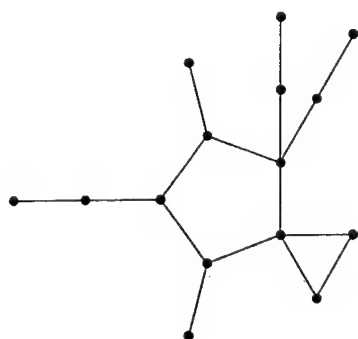
15.2.1



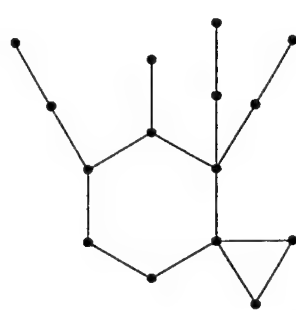
15.2.2



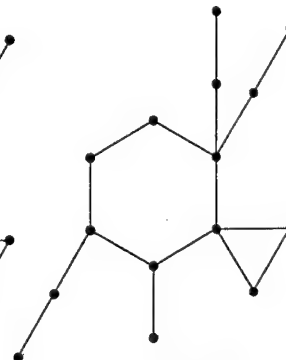
15.2.3



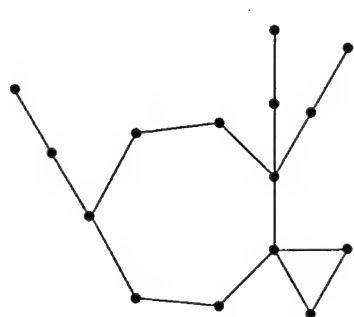
15.2.4



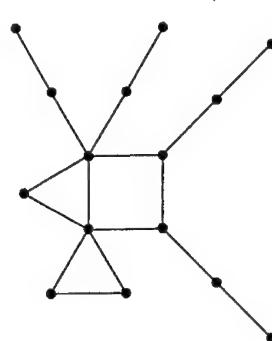
15.2.5



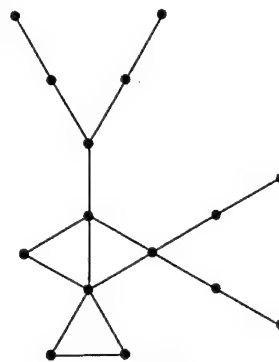
15.2.6



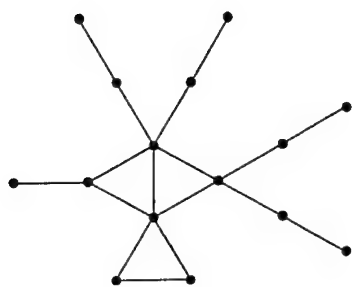
15.2.7



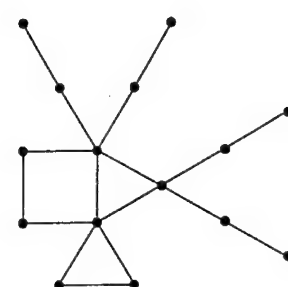
15.3.1



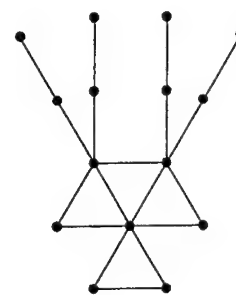
15.3.2



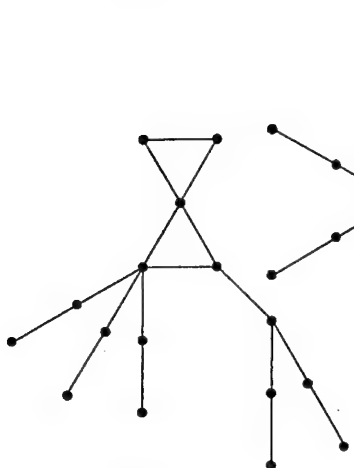
15.3.3



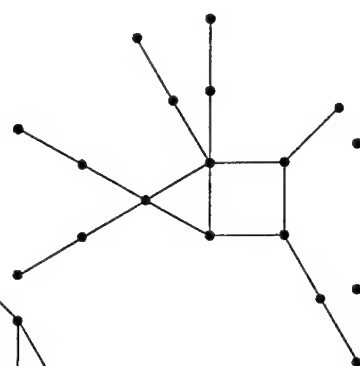
15.3.4



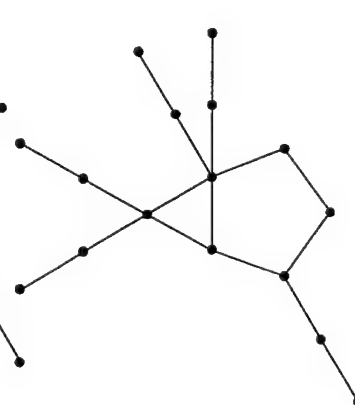
15.4.1



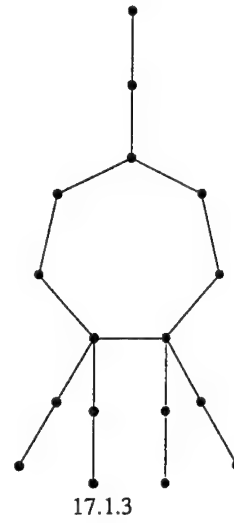
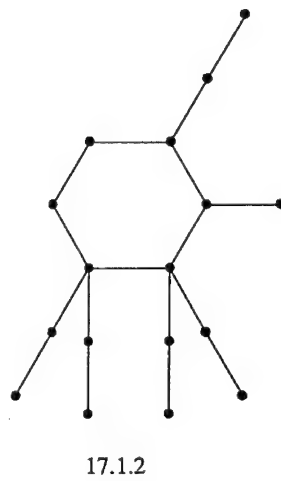
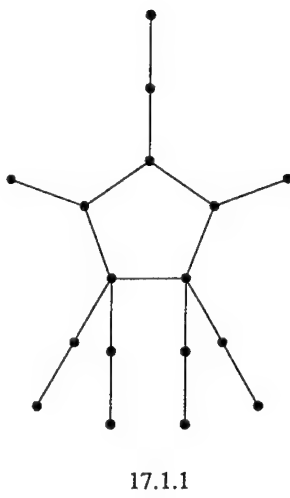
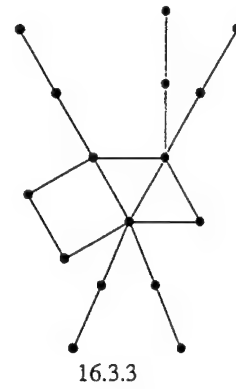
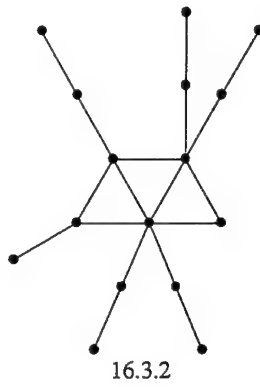
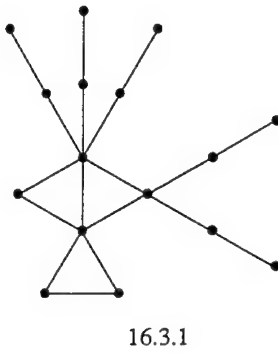
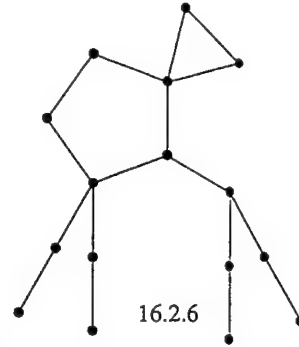
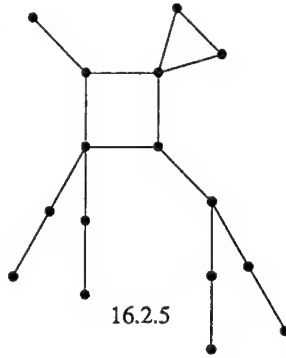
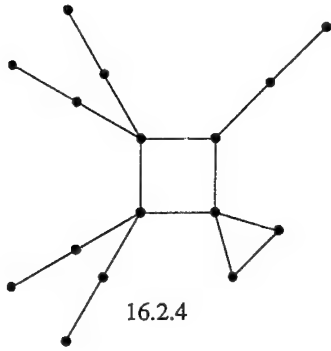
16.2.1

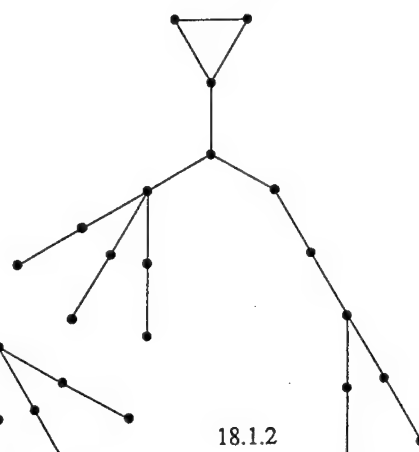
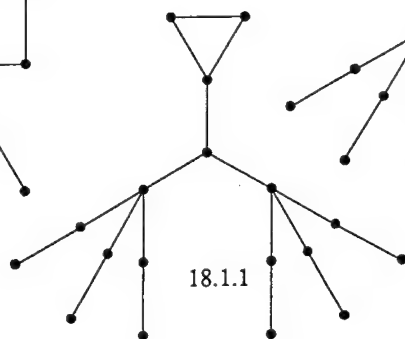
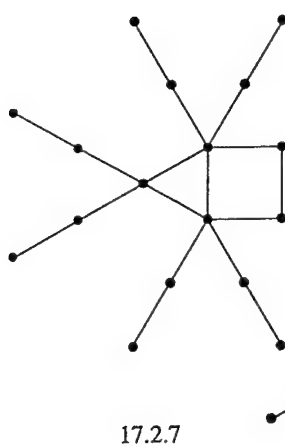
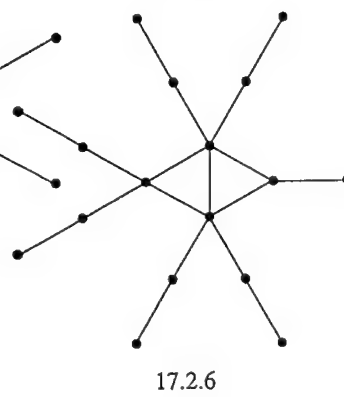
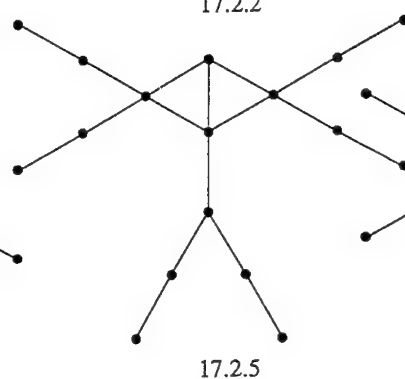
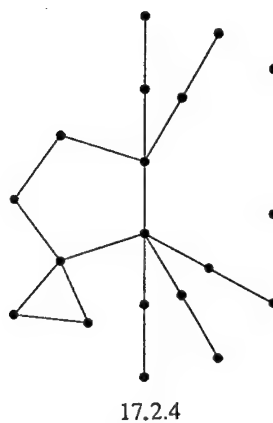
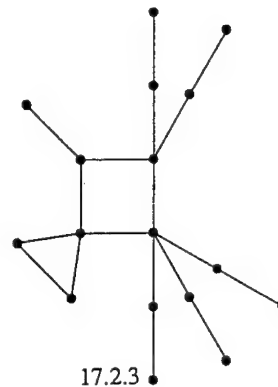
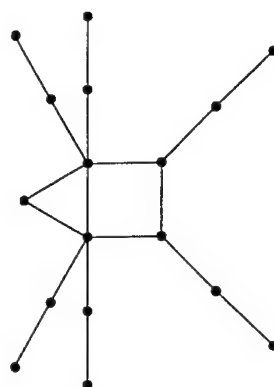
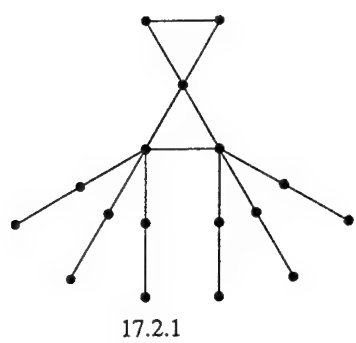


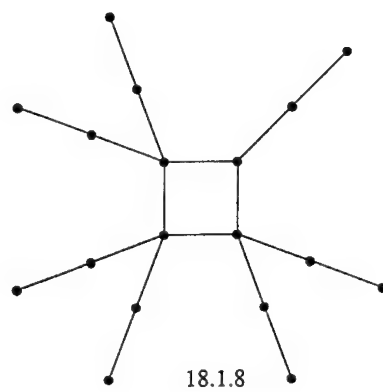
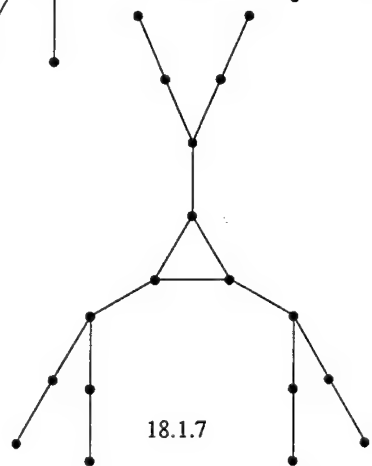
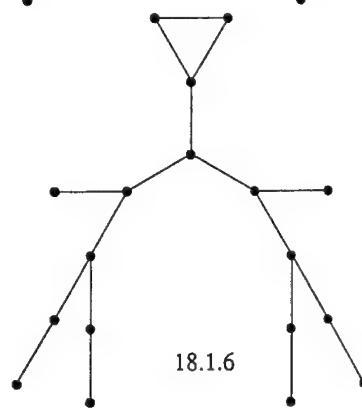
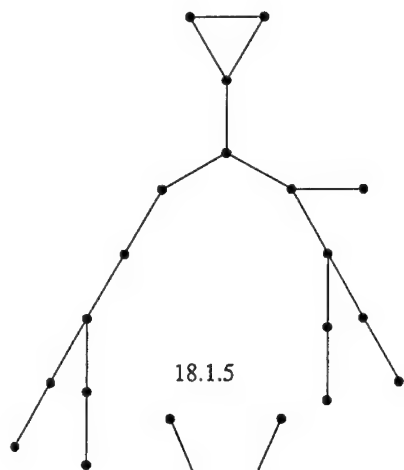
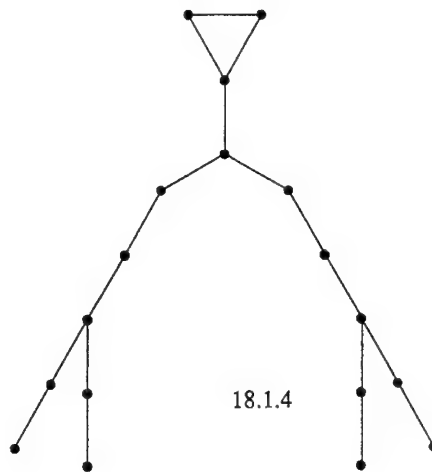
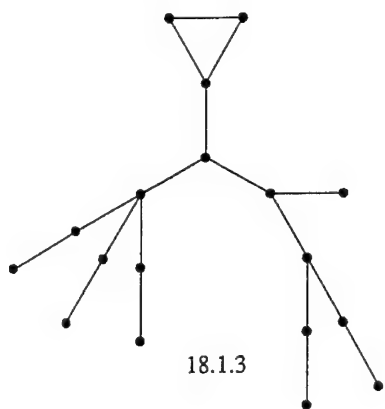
16.2.2

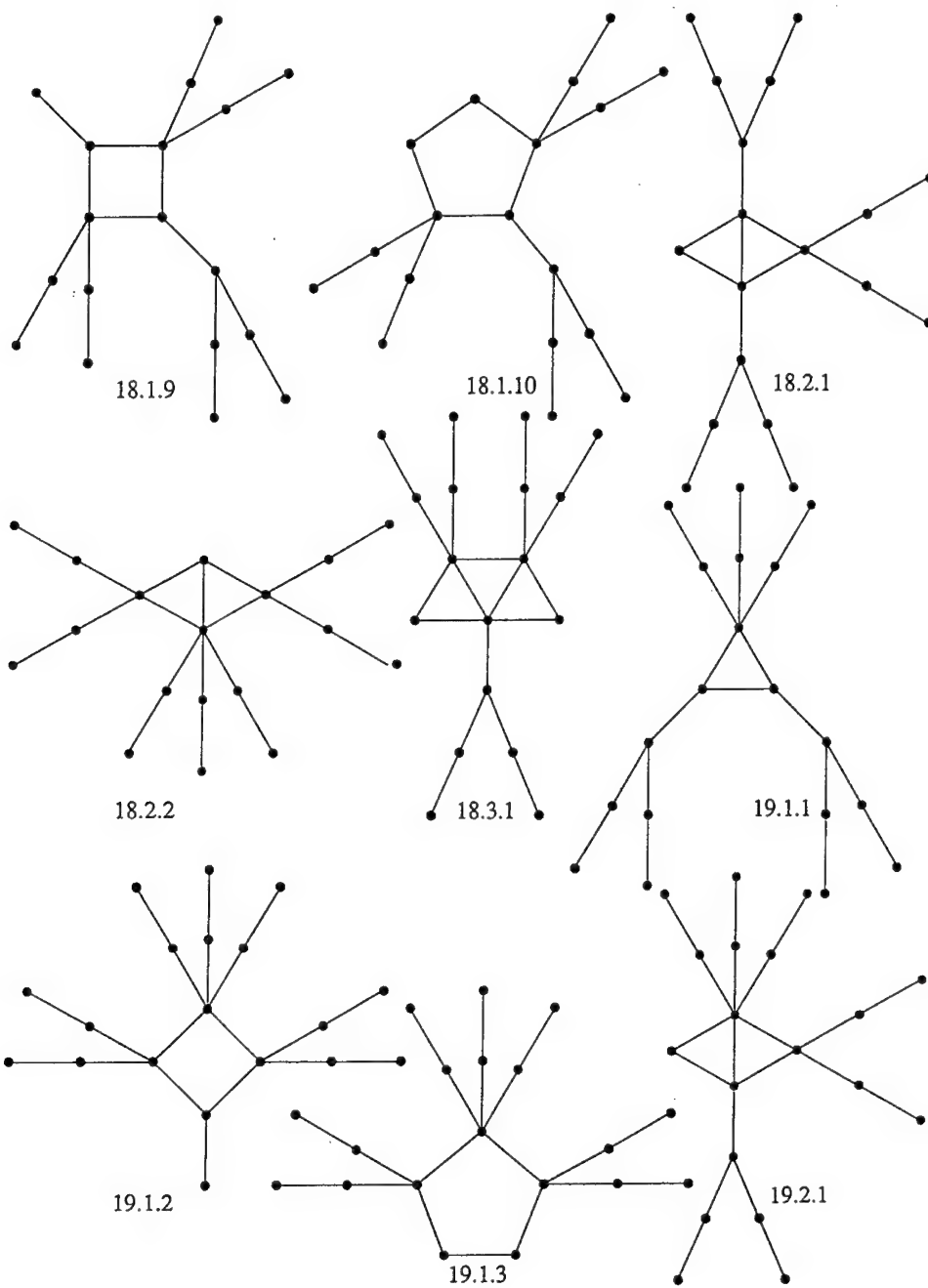


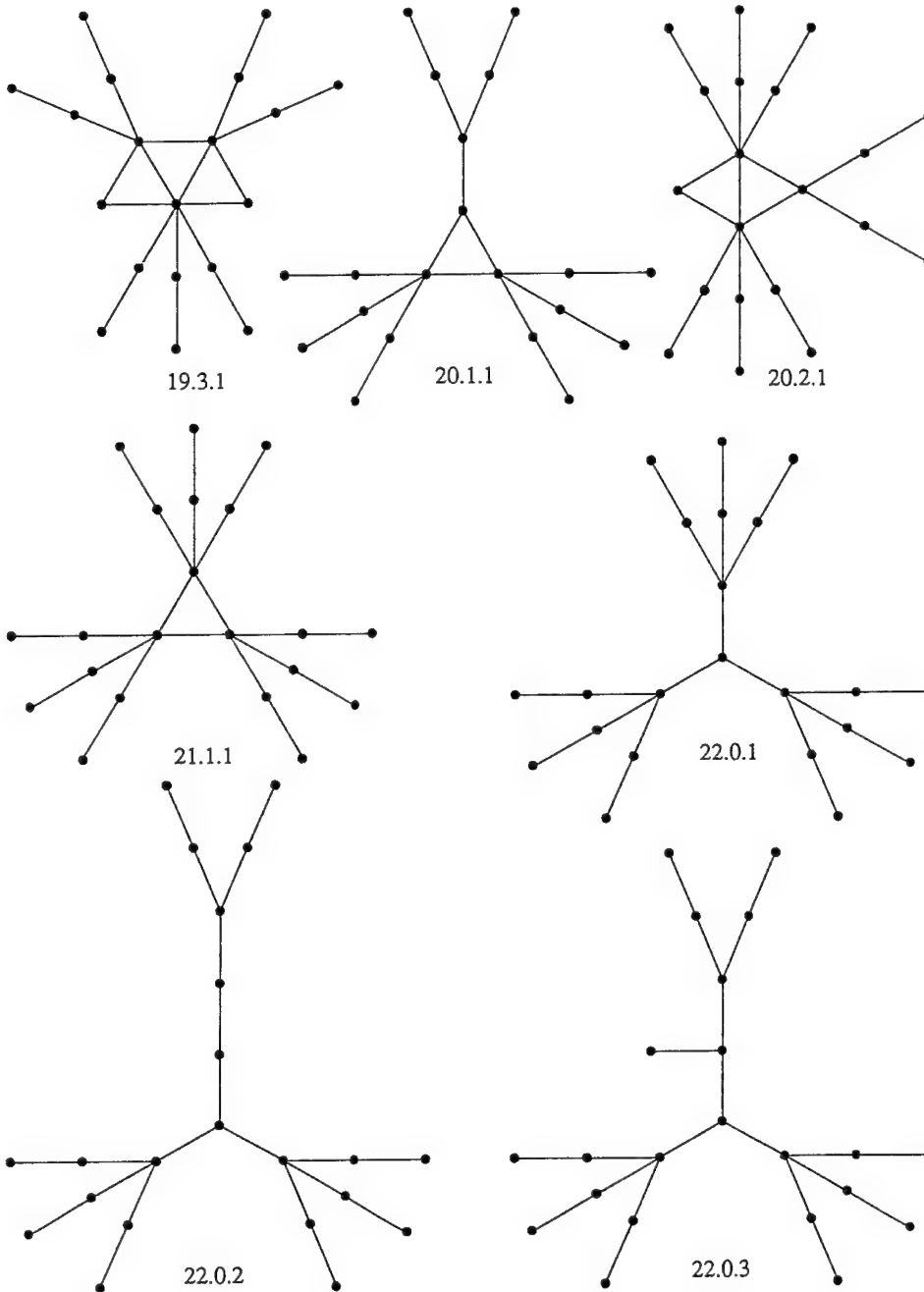
16.2.3

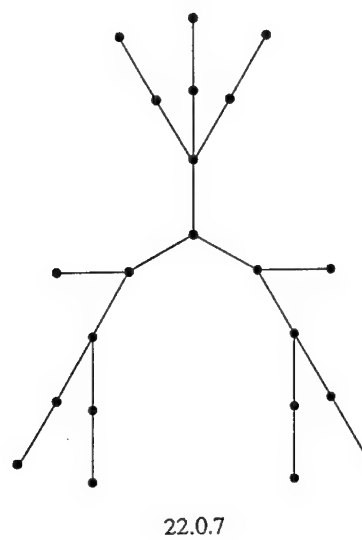
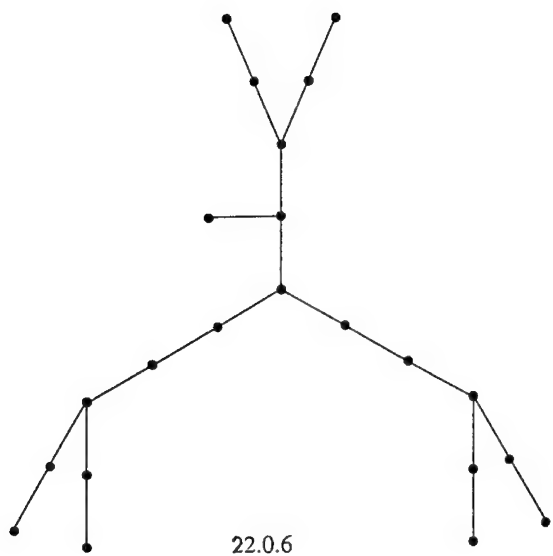
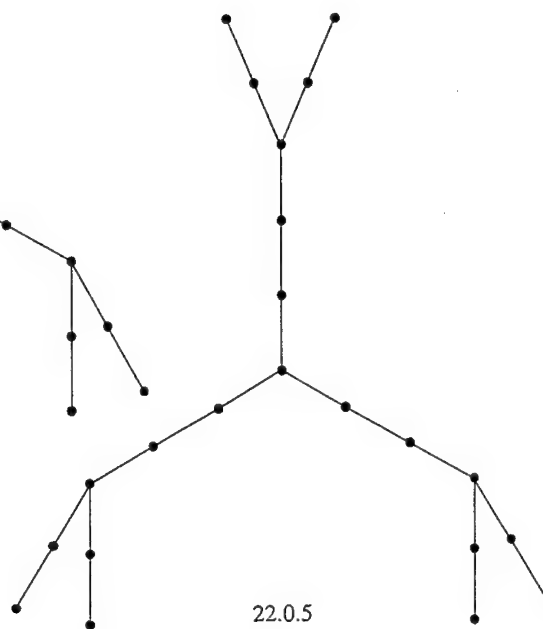
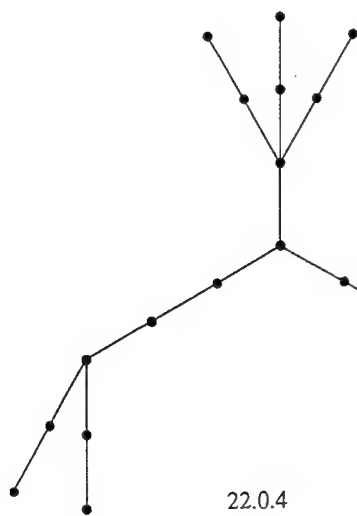


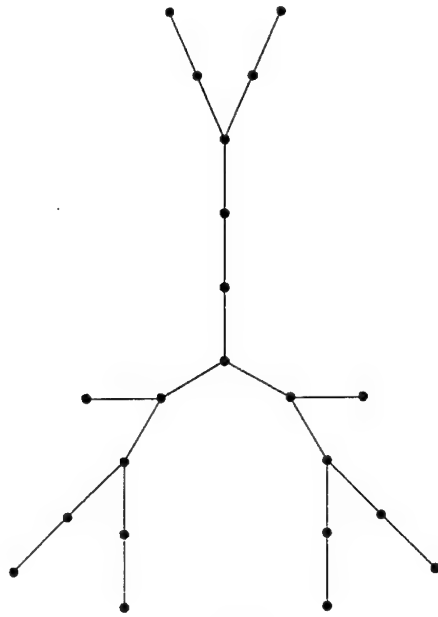




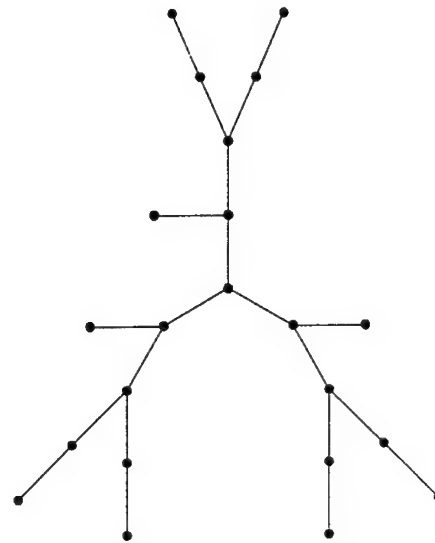




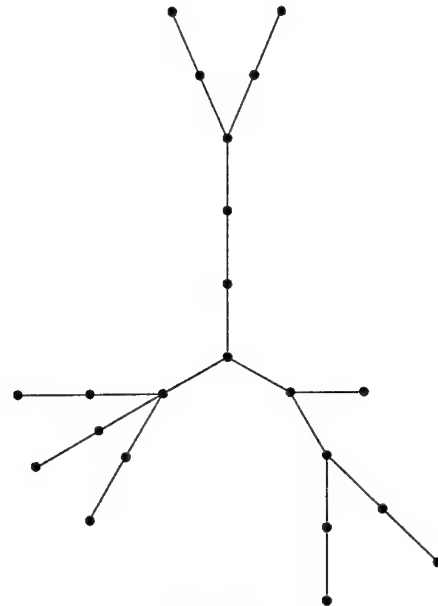




22.0.8



22.0.9



22.0.10

# Fast Stable Merging and Sorting in Constant Extra Space\*

B-C. HUANG<sup>1†</sup> AND M. A. LANGSTON<sup>2‡</sup>

<sup>1</sup>Department of Computer Science, University of South Carolina, Columbia, SC 29208.

<sup>2</sup>Department of Computer Science, University of Tennessee, Knoxville, TN 37996-1301.

*In an earlier research paper,<sup>9</sup> we presented a novel, yet straightforward linear-time algorithm for merging two sorted lists in a fixed amount of additional space. Constant of proportionality estimates and empirical testing reveal that this procedure is reasonably competitive with merge routines free to squander unbounded additional memory, making it particularly attractive whenever space is a critical resource. In this paper, we devise a relatively simple strategy by which this efficient merge can be made stable, and extend our results in a nontrivial way to the problem of stable sorting by merging. We also derive upper bounds on our algorithms' constants of proportionality, suggesting that in some environments (most notably external file processing) their modest run-time premiums may be more than offset by the dramatic space savings achieved.*

Received May 1989, revised August 1992

## 1. INTRODUCTION

It is a well-recognised phenomenon that no matter how much main memory (also known as core memory, directly-addressable memory, non-virtual memory or real memory) is made available to a collection of users and systems, it seems never to be enough to satisfy everyone completely. Although main memory is often rather cavalierly regarded as an inexpensive resource, its availability is in fact critical in many applications. We are reminded of the following passage by the witty and imaginative science writer D. E. H. Jones:<sup>13</sup>

Let's assume that the brain, like most computers, stores intelligence (programs) and memory (data) in the same form and distributed throughout the same volume. Then the more space is taken up by data the less is available for programs and working space. Clearly as life progresses and memories multiply, there must come a time when programs and working space get squeezed. This must be senility.

Naturally, main memory should be allocated and managed carefully to avoid thrashing<sup>2</sup> and other forms of *computer senility*. This is particularly true for heavily-used operations like merging and sorting, known to dominate a large portion of all available execution time over broad classes of computer systems.<sup>14</sup> This is even more evident when performing these operations over enormous external files. In such an environment, the overall processing time is frequently determined not by the speed of the algorithm used to process file segments internally, but rather by the ways in which available main memory can be used to accommodate more and larger buffers, thereby increasing device and channel parallelism while decreasing the number of I/O transfers required.

Unfortunately, for stable merging and sorting, the

obvious algorithms that work in asymptotically optimal time ( $O(n)$  and  $O(n \log n)$ , respectively) waste a whopping  $\Omega(n)$  extra memory cells for temporary storage. Conversely, the conspicuous ways to merge and sort in  $O(1)$  extra space are either unstable or require  $\Omega(n^2)$  time. A number of stable merging schemes that use more than linear time or more than constant extra space have been suggested,<sup>1, 3, 5, 6, 25</sup> as have several stable sorting strategies that use more than  $O(n \log n)$  time or more than  $O(1)$  extra space.<sup>7, 17, 19, 20</sup> Also, a routine that dynamically alters keys has been defined,<sup>12</sup> but is thus applicable only to files in which keys are explicitly stored within records. The only previously-known general method for stably merging and sorting in *both* optimal time and optimal extra space<sup>22, 23</sup> is widely regarded as a result of purely theoretical interest,<sup>15, 24</sup> since it is exceedingly complex and its time-complexity constant of proportionality is so huge that it hasn't even been derived. (Recent modifications have been suggested that simplify parts of this method, but its overall constant of proportionality remains prohibitively large and unbounded.)<sup>21</sup> This contrasts poorly with unstable merging, where much progress has been made in achieving practical and straightforward optimal time and space methods and with unstable sorting, where the simple heap-sort algorithm suffices.

The main result of this paper is a relatively simple, efficient and general scheme for stable merging (and thus stable merge-sorting) in optimal time and space. Our method is based on our recently-reported algorithm for fast, in-place unstable merging.<sup>9</sup> We also present an even more streamlined  $O(n \log n)$  time and  $O(1)$  extra space stable sorting procedure for files for which there is a reasonable limit on the number of times each key can appear. Significantly, and unlike previously-reported schemes to solve these problems, we derive explicit upper bounds on the number of key comparisons and record exchanges our methods require. Note that these strategies may be especially useful for operations such as stable polyphase, balanced or cascade merge-sorting with external storage media such as tape: larger initial runs mean fewer passes of the file (the common replacement-selection method is unstable), and more available memory for buffer space can mean less time consumed in each pass.

\* A preliminary version of a portion of this paper was presented at the International Conference on Computing and Information, held in Toronto, Ontario, Canada, in May 1989.

† This author's research has been supported in part by the Washington State University Graduate Research Assistantship Program.

‡ To whom correspondence should be addressed. This author's research has been supported in part by the National Science Foundation under grants ECS-8403859 and MIP-8603879; and by the Office of Naval Research under contract N00014-88-K-0343.

In the next section, we discuss pertinent background information and related work. Section 3 comprises the definitions and notational conventions we shall need to present and analyse our algorithms. In Section 4, we review the fundamental, optimal time and space unstable merge of Ref. 9 and define our modifications that ensure stability. Also, to provide an upper bound on the resultant procedure's worst-case constant of proportionality, we prove that the total number of key comparisons and record exchanges required never exceeds  $7n$  (plus lower-order terms). Section 5 extends our work to the problem of optimal time and space stable sorting in a nontrivial way. We devise an alternative to the obvious merge-sort strategy, and show that it never needs more than  $2.5 n \log_2 n$  (plus lower-order terms) key comparisons and record exchanges. In the final section, we draw a few conclusions from this effort and pose questions that we believe merit further investigation.

## 2. RELATED WORK

The general approach that we shall employ inherently relies on the notions of *internal buffering* and *block rearranging*, and can be traced back to the seminal work on unstable merging described in Ref. 16. Simply stated, with this approach we attempt to view a list of  $n$  records as a sequence of  $O(\sqrt{n})$  blocks, each of size  $O(\sqrt{n})$ . This allows us to employ one block as an internal buffer to aid in rearranging or otherwise manipulating the other blocks in constant extra space. Since only the contents of the buffer and the relative order of the blocks need be out of sequence, linear time is sufficient to perform a merge with the aid of selection sorting both the buffer and the blocks (each sort involves  $O(\sqrt{n})$  keys).

After the unstable method in Ref. 16 appeared, a stable procedure was proposed in Ref. 12 that, unfortunately, had the rather undesirable side-effect that records had to be alterable during its execution. Subsequently, a general algorithm for optimal time and space, stable merging and sorting was published,<sup>22,23</sup> as was a technique for simplifying parts of its control structure.<sup>21</sup> For the most part, however, these results have been of academic interest only, due primarily to their discouraging complexity and their prohibitively large time-complexity constants of proportionality.

More recent research efforts have begun to focus on simpler, more practical optimal time and space internal buffering and block rearranging strategies for unstable merging<sup>4,9,18</sup> as well as for extracting duplicates from a sorted list<sup>10</sup> and for all of the binary set and multiset operations on sorted lists,<sup>11</sup> with potential application to a number of file processing problems.

## 3. NOTATION, DEFINITIONS AND USEFUL SUBPROGRAMS

Let  $L$  denote a list (internal file) of  $n$  records, indexed from 1 to  $n$ . An algorithm for rearranging the order of the records of  $L$  is said to be *stable* if it ensures that, when it is done, records with identical keys retain the relative order they had before the algorithm began. We use  $KEY(i)$  as a shorthand to denote the key of the record with index  $i$ . Only the two common  $O(1)$  time and space primitive operations are assumed; namely, record exchanges and key comparisons. The exchange procedure,

$SWAP(i, j)$ , directs that the  $i$ th and  $j$ th records are to be exchanged. The comparison functions, for example  $KEY(i) < KEY(j)$ , return the expected Boolean values dependent on the relative values of the keys being compared.

From these primitive operations, we construct a few  $O(1)$  space useful subprograms for dealing with *blocks*. Let us define a block to be a set of records from  $L$  with consecutive indices. The *head* of a block is the record with the lowest index (or, informally, the 'leftmost' record in the block); the *tail* of a block is the record with the highest index (the 'rightmost' record in the block). The procedure  $BLOCKSWAP(i, j, h)$  exchanges a block of  $h$  records beginning at index  $i$  with a block of  $h$  records beginning at index  $j$  in  $O(h)$  time. We specify that blocks do not partially overlap (i.e., if  $i \neq j$  then  $h \leq |i - j|$ ) and that, when  $BLOCKSWAP$  is finished, records within a moved block retain the order they possessed before  $BLOCKSWAP$  was invoked. A block of  $h$  records beginning at index  $i$  is sorted in nondecreasing order by the procedure  $SORT(i, h)$ . The procedure  $BLOCKSORT(i, h, p)$  uses  $BLOCKSWAP$  to rearrange the  $p$  consecutive blocks, each with  $h$  records, beginning at index  $i$  so that their tails are sorted in nondecreasing order. To reduce unnecessary record movement, an important consideration when records are relatively long, we insist that  $BLOCKSORT$  use the  $O(p^2 + ph)$  time straight selection sort.<sup>14</sup>

The procedure  $ROTATE(i, h, l)$  rotates (circularly shifts) a block of  $h$  records, beginning at index  $i$ ,  $l$  places to the left. We assume that  $ROTATE$  is implemented in the common fashion with three sublist reversals, thereby requiring no more than  $h$  invocations of  $SWAP$ .

Finally, a pair of consecutive blocks, each sorted in nondecreasing order, is merged with  $BLOCKMERGE(i, h, k)$ , where the first block contains  $h$  records beginning at index  $i$  and the second contains  $k$  records beginning at index  $i + h$ .  $BLOCKMERGE$  uses  $ROTATE$  to merge the shorter block into the longer one. For example, if  $h \leq k$ , then  $BLOCKMERGE$  merges the first block *forward* into the second as follows. A binary search of the second block is used to find the leftmost *insertion point* for the leftmost record of the first block. That is, assuming  $KEY(i + h) < KEY(i) \leq KEY(i + h + k - 1)$ , the displacement  $p$  is computed for which  $KEY(i + h + p) < KEY(i) \leq KEY(i + h + p + 1)$ , followed by an invocation of  $ROTATE(i, h + p, h)$ . The first record of the shorter block and all records to its left are now merged. The merge is completed by iterating this operation until one of the blocks is exhausted, resulting in a time complexity of  $O(h^2 + k)$ . (There are at most  $O(h \log k)$  comparisons. Records from the shorter block are moved no more than  $h$  times, while records from the longer block are moved only once.) Of course, if  $h > k$ , then  $BLOCKMERGE$  is better off to merge the second block *backward* into the first in  $O(h + k^2)$  time.

## 4. STABLE IN-PLACE MERGING

### 4.1 A Review of the Fundamental, Unstable Merge

Suppose  $L$  contains two sublists to be merged, each with its keys in nondecreasing order. In Ref. 9 we presented a fast and surprisingly simple algorithm for (unstable) merging in linear time and constant extra space. Even without 'tinkering' with it to achieve an especially

efficient implementation, its average run time on large lists exceeds that of the standard, widely-used merge (which is free to exploit  $O(n)$  temporary extra memory cells) by less than a factor of two. Aspects that contribute to its straightforwardness include a rearrangement of blocks before a merging phase is initiated and an efficient pass of the internal buffer across the list to reduce unnecessary record movement.

We now briefly review the central features of this  $O(n)$  time and  $O(1)$  extra space method, with a number of simplifying assumptions made about  $L$  to facilitate discussion. We refer the reader to Ref. 9 for a complete exposition of the algorithm, an example, and the  $O(\sqrt{n})$  time and  $O(1)$  space implementation details necessary for handling arbitrary inputs.

Let us suppose that  $n$  is a perfect square, and that we have already permuted the records of  $L$  so that  $\sqrt{n}$  largest-keyed records are at the front of the list (their relative order there is immaterial), followed by the remainders of the two sublists, each of which we now assume contains an integral multiple of  $\sqrt{n}$  records in nondecreasing order.

Therefore, we view  $L$  as a series of  $\sqrt{n}$  blocks, each of size  $\sqrt{n}$ . We will use the leading block as an internal buffer to aid in the merge. Our first step is to invoke *BLOCKSORT* on the  $\sqrt{n}-1$  rightmost blocks, after which their tails form a nondecreasing key sequence. (In this setting, selection sort requires only  $O(n)$  key comparisons and record exchanges.) Records within a block retain their original relative order.

Next, we locate two series of records to be merged. The first series begins with the head of block 2 and terminates with the tail of block  $i$ ,  $i \geq 2$ , where block  $i$  is the first block such that the key of the tail of block  $i$  exceeds the key of the head of block  $i+1$ . The second series consists solely of the records of block  $i+1$ . We now use the buffer to merge these two series. That is, we repeatedly compare the leftmost unmerged record in the first series to the leftmost unmerged record in the second, swapping the smaller-keyed record with the leftmost buffer element. Ties are broken in favour of the leftmost series. (In general, the buffer may be broken into two pieces as we merge.) We halt this process when the tail of block  $i$  has been moved to its final position.

We now locate the next two series of records to be merged. This time, the first begins with the leftmost unmerged record of block  $i+1$  and terminates as before for some  $j \geq i$ . The second consists solely of the records of block  $j+1$ . We resume the merge until the tail of block  $j$  has been moved.

We continue this process of locating series of records and merging them until we reach a point where only one such series exists, which we merely shift left, leaving the buffer in the last block. A sort of the buffer completes the merge of  $L$ .

$O(1)$  space suffices for this procedure, since the buffer was internal to the list, and since only a handful of additional pointers and counters are necessary.  $O(n)$  time suffices as well, since the block sorting, the series merging and the buffer sorting each require at most linear time.

## 4.2 Obstacles to Stability

The primary problem to be addressed in order to

achieve stability is the need to be able to distinguish blocks as to whether each originated in the first or the second sublist. This is a more difficult task than it may seem at first blush. A number of schemes will do if each block has different keys at its head and its tail. For example, we could simply make a temporary swap of the records at the head and tail of a block if and only if it originated in, say, the second sublist. (Such a swap would be made during the blocking-sorting phase and undone during the series-merging phase.) The real problem lies with *homogeneous* blocks, those in which every record in the block has the same key as every other record in the block. To illustrate this conundrum, suppose we know by some artifice that block  $i > 2$  originated in the first sublist, but only that block  $i-1$  is homogeneous. Also, suppose the key of the tail of block  $i-1$  equals the key of the head of block  $i$ , but is strictly less than the key of the tail of block  $i$ . In this circumstance, stability is jeopardized since we cannot determine whether the head of block  $i$  should be merged to the left or to the right of the records of block  $i-1$ . (It should go to the left if block  $i-1$  originated in the second sublist, but to the right otherwise.)

Additionally, we must be wary of a few other details that, if neglected, can compromise stability. For example, we need to load the buffer with records having distinct keys, if that is possible, since the buffer's contents are arbitrarily permuted during the series-merging phase. Correspondingly, we must provide for the special case in which there are not enough distinct keys to fill the buffer. We also want to make the *BLOCKSORT* subprogram of the block-sorting phase stable, because otherwise a large collection of homogeneous blocks may be unpredictably rearranged. Finally, as with the fundamental, unstable merge,<sup>9</sup> we need to specify implementation details for handling lists and sublists of arbitrary sizes.

## 4.3 The Main Idea

Since the possibility of troublesome homogeneous blocks prevents the use of any simple scheme for identifying individual blocks as to their origin, we shall seek instead to devise a strategy by which we can distinguish a *series* of consecutive blocks from the first sublist from a *series* of consecutive blocks from the second. To this end, it is enough if we can be sure of the first and last block in every series from the second sublist only.

We shall encode this information in  $L$  during the (stable) block-sorting phase and decode it during the series-merging phase. To encode, we use two memory cells, one to point to the (post-sorting) position of the leftmost block of the leftmost second-sublist series and one to point to the (post-sorting) position of the rightmost block of the rightmost second-sublist series. As the sort phase progresses, we mark each series by exchanging the head of the rightmost block of one second-sublist series with the tail of the leftmost block of the next second-sublist series.

We thus make use of the fact that one or more blocks from the first sublist must lie between the blocks we have modified, insuring that we can correctly decode the series delimiters during the series-merging phase. That is, it directly follows that the head of the rightmost block of a second-sublist series will temporarily have a key strictly

greater than that of the record to its immediate right, while the tail of the leftmost block of a second-sublist series will temporarily have a key strictly less than that of the tail of the block to its immediate left. Of course, with this stringent mechanism for defining each distinct series to be merged, we do not employ the simpler criteria used in the unstable merge to locate series. Now, as we merge, we undo the exchanges that delimit the series and always break ties in favour of the series from the first sublist.  $O(\sqrt{n})$  time and  $O(1)$  space are sufficient for this scheme.

#### 4.4 Other Relevant Details

We attempt to load the internal buffer with distinct-keyed records as follows. We begin at the right end of the first sublist and scan to the left. When a comparison of adjacent keys reveals that the leftmost copy of a key has been found, that record is coalesced into the buffer. Other records are exchanged with the rightmost current buffer element. Therefore, the buffer begins with size zero and grows as we 'roll' it to the left. When it has attained size  $\sqrt{n}$ , we invoke *ROTATE* to left-justify it. At the end of the series-merging phase (the buffer is now right-justified), we stably merge the buffer with the remainder of the list with a backward *BLOCKMERGE* using leftmost insertion points.

In the event that we exhaust the first sublist without filling the internal buffer, we must employ fewer but larger blocks. Specifically, if we obtain only  $s < \sqrt{n}$  buffer elements, then we use  $s$  blocks, each of size at most  $\lceil n/s \rceil$ . Although this permits the use of the stable *BLOCKSORT* described below, it is of no help in the merging phase. Fortunately, however, such a small number of distinct keys in the first sublist ensures that we can, in  $O(n)$  time, stably merge the sorted series of blocks with a left-to-right series of backward *BLOCKMERGE* operations, each using the proper insertion point, which is the leftmost if the left series is from the second sublist, and the rightmost otherwise. Since the buffer does not in this scheme end up adjacent to the unmerged suffix of the right series when the left is exhausted, we use a pointer to indicate this boundary, namely, the location of the leftmost record in the right series not moved by *ROTATE*. (Although this method is easy to implement, it is not perhaps obvious that it takes only linear time. See Section 4.5.) We then stably merge the buffer with the remainder of the list with a forward *BLOCKMERGE* using leftmost insertion points.

Our *BLOCKSORT* implementation must be stable. This is easily achieved by first invoking *SORT* on the buffer and then using it to 'remember' the original block sequence. That is, we exchange each buffer element with the proper block's tail before blocks are rearranged, and then undo each exchange as the corresponding block is selected by *BLOCKSORT*. This simple scheme, a variation of the 'segment insertion process' used in Ref. 23, thus restores the tails in time to perform the series-encoding task (as described in Section 4.3) as the sort progresses.

Finally, for lists and sublists of arbitrary sizes, we employ a method analogous to the one we used for unstable merging.<sup>9</sup> This gives potential rise to one small block (of size less than  $\lceil \sqrt{n} \rceil$ ) at the extreme right end of the list, and one at the left end to the immediate right of the buffer. For the right block, no modification is

necessary. For the left one, we observe that in general a *ROTATE* may be necessary to insert the block in its proper place, after *BLOCKSORT* is finished, when a second-sublist series should precede it.

#### 4.5 Constant of Proportionality Bounds

In an effort to measure the practical potential of this stable, optimal time and space merge, we shall study the number of key comparisons and record exchanges it demands. These two primitives are generally regarded as by far the most time consuming operations for internal file processing, requiring storage-to-storage instructions for many architectures. Since it is possible to count them independently from the code of any particular implementation, their total gives a meaningful estimate of the size of the linear-time constant of proportionality for the algorithm we have devised. (As for the issue of constant extra space, a careful review of our method reveals that a couple of dozen additional storage cells is all we need for use as pointers and counters.)

We now proceed to derive a worst-case bound on the key-comparison and record exchange sum. For simplicity, we allow for a (possibly unrealizable) worst-case scenario, implying that the figures we produce may be rather conservative upper bounds. (This is offset to some extent, especially for small inputs, by the fact that we are ignoring operations bounded above by lower-order terms. For example, sorting the buffer can be accomplished in-place with heap-sort in  $O(\sqrt{n} \log n)$  time. In fact, our main idea for achieving stability needs only  $O(\sqrt{n})$  time.) Let  $n_1(n_2)$  denote the size of the first (second) sublist, and thus  $n_1 + n_2 = n$ .

Consider the general case, in which there are plenty of distinct keys to fill the buffer. Extracting the buffer uses at most  $n_1$  comparisons and  $n_1$  exchanges. By selecting blocks from right to left, our stable *BLOCKSORT* requires at most  $\sum_{i=1}^{n_1} i < n_2/2$  comparisons (the first sublist does not become disordered) and fewer than  $n$  exchanges (each of the  $\sqrt{n}-1$  *BLOCKSWAP* invocations puts  $\sqrt{n}$  records in position). For arbitrary list and sublist sizes, the *ROTATE* used to move the left small block needs at most  $n_2$  exchanges. For the series-merging phase, fewer than  $n$  comparisons and  $n$  exchanges suffice. Finally, since we use a binary search to locate the insertion points for merging back the buffer, this operation needs only  $O(\sqrt{n} \log n)$  comparisons and no more than  $(n - \sqrt{n}) + \sum_{i=1}^{n_1} i < 1.5n$  exchanges. Therefore, we are guaranteed a worst-case key-comparison and record-exchange grand total of something less than  $6.5n$ .

For the special case in which we exhaust the first sublist before filling the buffer, the only operation whose constant is affected is the series-merge, which is implemented with a series of *BLOCKMERGE* invocations. In this simple scheme, we work from left to right, always merging a series of one or more blocks with the single block to its immediate right. Since there are only  $s < \sqrt{n}$  distinct keys in the first sublist, we shall in this case employ two binary searches to locate insertion points for merging (the first search on the series or block from the first sublist, the second search on the other) and require only  $O(\sqrt{n} \log n)$  comparisons. As for exchanges, we observe that no record is moved to the right more than once. Each distinct key in the first sublist gives rise to at

most one invocation of *ROTATE*, except when such a key is represented in two distinct first-sublist series (it cannot be in three or more), which can happen at most  $s/2$  times, each time giving rise to at most one more *ROTATE* operation. Since each *ROTATE* moves at most  $n/s$  records to the left, a total of at most  $n + 1.5s(n/s) = 2.5n$  exchanges are required. Hence, we are assured a worst-case key-comparison and record-exchange grand total bounded above by  $7n$ .

For comparison, consider previously-published methods to solve this problem.<sup>21,23</sup> Curiously, these works focus only on establishing the existence of algorithms, and include no constant of proportionality analysis. However, we have studied the intricate details of the general method they use, as described in full in,<sup>22</sup> and have found that they yield a worst-case key-comparison and record-exchange grand total in excess of  $15n$ . Perhaps more importantly, we observe that our approach is dramatically simpler. As one rough estimate of the cost of stability, we remark that the key-comparison and record-exchange total of our underlying, unstable merge was bounded above by  $3.5n$  in Ref 9.

## 5. STABLE IN-PLACE SORTING

### 5.1 The Direct Merge-Sort Approach

We can, naturally, now take the simple course suggested in Ref. 23 and directly use our stable, in-place linear-time merge as a subroutine for merge-sorting. We observe that this gives rise to a key-comparison and record-exchange total bounded above by  $7n \log_2 n$ , plus lower order terms (elementary combinatorics guarantees that our merge's sublinear terms, the largest of which is  $O(\sqrt{n \log n})$ , give rise to terms of at most  $O(n)$  in the resulting 'divide and conquer' merge-sort scheme).

As with the traditional, memory-dependent merge-sort, this scheme will be more effective in practice if we use a less-complicated, quadratic-time sort when subfile sizes fall below some established 'break-even' point that depends on a number of factors local to a given sorting environment. Even so, a lot of time will be spent in extracting an internal buffer at each call of the merge subroutine. We shall demonstrate in the next subsection that this effort can be avoided as long as no single key is permitted to dominate the file.

### 5.2 A Nontrivial Sort-by-Merging Strategy

Consider an environment in which no key is duplicated more than about  $\sqrt{n}$  times, which may be plausible in many settings. With this restriction, we now outline a method to sort stably by merging so as to bypass much of the overhead involved in a direct merge-sorting scheme. (Nevertheless, without this restriction, we must endorse instead the direct merge-sort approach. That is, we have found no general mechanism by which our nontrivial merge-sort described below can stably handle large numbers of homogeneous blocks in its later passes without overstepping our professed goal of presenting relatively simple and practical algorithms.)

To facilitate discussion, suppose that  $n$  is of the form  $2^{2k} + 2^k$  for some positive integer  $k$ . We assume that no single key is represented more than  $2^k$  times. Consequently, we can use blocks of size  $2^k$  since there are more than  $2^k$  distinct keys available for the buffer. (No

laborious discussion of implementation details is necessary. If  $n$  is not of the proper form, we merely determine the value of  $k$  for which  $2^{2k} + 2^k < n < 2^{2(k+1)} + 2^{k+1}$ . Our restriction becomes that no single key is represented more than  $2^{k-1}$  times, insuring blocks of size  $2^{k+1}$ , which will do.) Figure 1a) depicts such a list with  $k = 2$  and  $n = 20$ . Only record keys are listed, denoted by capital letters. Subscripts are included to keep track of duplicate keys as the algorithm progresses.

The first step of the algorithm is to fill an internal buffer of size  $2^k$  with records having distinct keys. Thus we seek to convert  $L$  into the form  $BA$ , where  $B$  is the buffer and  $A = L - B$  such that  $STABLESORT(L) = STABLEMERGE(B, STABLESORT(A))$ . To do this, we perform a left-to-right scan of  $L$ , 'growing'  $B$  as a sorted sublist. The first record of  $L$  is placed in  $B$ . As we scan the  $i$ th record,  $i > 1$ , we conduct a binary search on  $B$  to see if its key is already present. If so, we go on to scan the next record. If not, we *ROTATE* the appropriate segment of  $L$  so that  $B$ 's rightmost record occupies position  $i-1$ . We then insert the new record into  $B$ . As soon as  $B$  is filled, we invoke *ROTATE* to make  $B$  a prefix of  $L$ . Fig. 1 illustrates how this process modifies our example list of 20 elements. We have used only  $O(1)$  extra space,  $O(n)$  exchanges and  $O(n \log 2^k = nk)$  comparisons.

$G_1 E_1 G_2 B_1 E_2 E_3 D_1 D_2 A_1 C_1 A_2 E_4 H_1 C_2 D_3 H_2 B_2 F_1 A_3 C_3$

a) Example list  $L$ , with  $k = 2$  and  $n = 20$ .

$\underbrace{E_1 G_1}_{B} G_2 B_1 E_2 E_3 D_1 D_2 A_1 C_1 A_2 E_4 H_1 C_2 D_3 H_2 B_2 F_1 A_3 C_3$

b) First two buffer elements are found.

$G_2 B_1 \underbrace{E_1 G_1}_{B} E_2 E_3 D_1 D_2 A_1 C_1 A_2 E_4 H_1 C_2 D_3 H_2 B_2 F_1 A_3 C_3$

c) Third buffer element is found.

$G_2 E_2 E_3 \underbrace{B_1 D_1 E_1 G_1}_{B} D_2 A_1 C_1 A_2 E_4 H_1 C_2 D_3 H_2 B_2 F_1 A_3 C_3$

d) Buffer is filled.

$\underbrace{B_1 D_1 E_1 G_1}_{B} \underbrace{G_2 E_2 E_3 D_2 A_1 C_1 A_2 E_4 H_1 C_2 D_3 H_2 B_2 F_1 A_3 C_3}_{A}$

e) Buffer is repositioned.

Figure 1. Filling the internal buffer,  $B$ .

In the second step of the algorithm, we use  $B$  to conduct the first  $k+1$  passes of a merging sort. We first employ the rightmost buffer element to conduct a left-to-right pass of  $A$ , producing a sequence of sorted two record sublists as we go. The second merging pass is done with the rightmost two remaining buffer elements, thus time producing sorted four-element sublists, and so on. The size of  $B$  ensures that this simple strategy suffices for  $k$  passes, each doubling the length of the sorted sublist (Since  $2^k = \sum_{i=1}^k 2^{i-1} + 1$ , pass  $i$  is performed with  $2^i$  buffer elements for  $1 \leq i < k$ , but in pass  $k$  we use  $2^{k-1} + 1$  elements, one more than we really need.) No that  $B$  is reassembled as a suffix of  $L$ , we proceed to use it in a right-to-left fashion to perform merging pass  $k+1$ .

Therefore  $BA$  has been transformed into  $BC$ , where  $C$  contains  $2^{k-1}$  sorted sublists, each of size  $2^{k-1}$ . See Figure 2. No more than  $O(1)$  space and  $O(nk)$  time has been used.

$B_1 D_1 E_1 G_1 \quad G_2 \quad E_2 \quad E_3 \quad D_2 \quad A_1 \quad C_1 \quad A_2 \quad E_4 \quad H_1 \quad C_2 \quad D_3 \quad H_2 \quad B_2 \quad F_1 \quad A_3 \quad C_3$   
 $\underbrace{\hspace{1.5cm}}_B$

a) Example list after buffer is filled.

$B_1 \quad D_1 \quad E_1 \quad \underline{E_2 G_2} \quad \underline{D_2 E_3} \quad \underline{A_1 C_1} \quad \underline{A_2 E_4} \quad \underline{C_2 H_1} \quad \underline{D_3 H_2} \quad \underline{B_2 F_1} \quad \underline{A_3 C_3} \quad G_1$

b) First pass is performed.

$\underline{D_2 E_3 E_3 G_2} \quad \underline{A_1 A_2 C_1 E_4} \quad \underline{C_2 D_3 H_1 H_2} \quad \underline{A_3 B_2 C_3 F_1} \quad \underline{B_1 D_1 E_1 G_1}$   
 $\underbrace{\hspace{1.5cm}}_B$

c) Second pass is performed.

$\underline{E_1 D_1 B_1 G_1} \quad \underline{A_1 A_2 C_1 D_2 E_2 E_3 E_4 G_2 A_3 B_2 C_2 C_3 D_3 F_1 H_1 H_2}$   
 $\underbrace{\hspace{1.5cm}}_B \quad \underbrace{\hspace{1.5cm}}_C$

d) Pass  $k+1=3$  is performed.

Figure 2. The first  $k+1$  merging passes.

For the third step of the algorithm, it is helpful to think of  $C$  as a collection of  $2^k$  sorted blocks, each of size  $2^k$ . In pass  $k+2$ , we use  $B$  to obtain  $2^{k+2}$  sublists, each of size  $2^{k+2}$  as follows. Let  $X$  and  $Y$  denote a pair of sublists in  $C$  to be merged. We first locate the block of  $X$  whose head contains the smallest key in  $X$ . Let  $X_1$  denote this block. Let  $Y_1$  denote the corresponding block of  $Y$ . (Note: We will search  $X$  and  $Y$  for these and all remaining merging blocks as they are needed. Although blocks will always be sorted internally, they will in general become unordered within a sublist with respect to each other. This turns out to be advantageous in the long run, requiring but a single *BLOCKSORT* after the final pass rather than a series of *BLOCKSORT*'s at each pass that would result in a great deal of unnecessary record movement.)

$X_1$  and  $Y_1$  are now merged into the buffer's block until it is filled. Whenever a block is filled, we must determine the next block to fill, as follows. If the buffer is now contained within one block, then that block is filled next. Otherwise, the buffer must be split into two pieces, one in a block of  $X$  and the other in a block of  $Y$ . If one piece is a suffix for its block (both cannot be), then we resume the merge at that block. If not, then each piece must be a prefix for its block, and we resume the merge at the block with the smaller tail, ties broken in favour of the  $X$  block. After all blocks of  $X$  and  $Y$  are merged in this manner, the buffer (now in one block) is moved back to its original position and we begin to merge the next pair of sublists in the same fashion.

This procedure is repeated in every subsequent pass, each time with half as many sublists, each sublist with twice as many blocks, until pass  $2k$ , which is the last. See Figure 3. In this step, we have used at most constant extra space and each of the  $k-1$  passes needs only linear time.

$\underline{E_1 D_1 B_1 G_1} \quad \underline{A_1 A_2 C_1 D_2} \quad \underline{E_2 E_3 E_4 G_2} \quad \underline{A_3 B_2 C_2 C_3} \quad \underline{D_3 F_1 H_1 H_2}$   
 $\underbrace{\hspace{1.5cm}}_B$

a) Example list after pass  $k+1=3$ .

$\underline{A_1 A_2 A_3 B_2} \quad \underline{\underline{E_1 D_1 C_1 D_2}} \quad \underline{E_2 E_3 E_4 G_2} \quad \underline{B_1 G_1 C_2 C_3} \quad \underline{D_3 F_1 H_1 H_2}$   
 $\underbrace{\hspace{1.5cm}}_{\text{block 1}}$

b) First block is merged (buffer elements underscored).

$\underline{A_1 A_2 A_3 B_2} \quad \underline{\underline{E_1 D_1 B_1 G_1}} \quad \underline{E_2 E_3 E_4 G_2} \quad \underline{C_1 C_2 C_3 D_2} \quad \underline{D_3 F_1 H_1 H_2}$   
 $\underbrace{\hspace{1.5cm}}_{\text{block 1}} \quad \underbrace{\hspace{1.5cm}}_{\text{block 2}}$

c) Second block is merged (buffer elements underscored).

$\underline{A_1 A_2 A_3 B_2} \quad \underline{D_3 E_2 E_3 E_4} \quad \underline{\underline{D_1 B_1 G_1 G_2}} \quad \underline{C_1 C_2 C_3 D_2} \quad \underline{E_1 F_1 H_1 H_2}$   
 $\underbrace{\hspace{1.5cm}}_{\text{block 1}} \quad \underbrace{\hspace{1.5cm}}_{\text{block 3}} \quad \underbrace{\hspace{1.5cm}}_{\text{block 2}} \quad \underbrace{\hspace{1.5cm}}_{\text{block 2}} \quad \underbrace{\hspace{1.5cm}}_{\text{block 2}}$

d) Third block is merged (buffer elements underscored).

$\underline{A_1 A_2 A_3 B_2} \quad \underline{D_3 E_2 E_3 E_4} \quad \underline{F_1 G_2 H_1 H_2} \quad \underline{C_1 C_2 C_3 D_2} \quad \underline{\underline{E_1 D_1 G_1 B_1}}$   
 $\underbrace{\hspace{1.5cm}}_{\text{block 1}} \quad \underbrace{\hspace{1.5cm}}_{\text{block 3}} \quad \underbrace{\hspace{1.5cm}}_{\text{block 4}} \quad \underbrace{\hspace{1.5cm}}_{\text{block 2}} \quad \underbrace{\hspace{1.5cm}}_B$

e) Fourth block is merged.

$\underline{E_1 D_1 G_1 B_1} \quad \underline{D_3 E_2 E_3 E_4} \quad \underline{F_1 G_2 H_1 H_2} \quad \underline{C_1 C_2 C_3 D_2} \quad \underline{A_1 A_2 A_3 B_2}$   
 $\underbrace{\hspace{1.5cm}}_B \quad \underbrace{\hspace{1.5cm}}_{\text{block 3}} \quad \underbrace{\hspace{1.5cm}}_{\text{block 4}} \quad \underbrace{\hspace{1.5cm}}_{\text{block 2}} \quad \underbrace{\hspace{1.5cm}}_{\text{block 1}}$

f) Buffer is repositioned.

Figure 3. Merging pass  $k+2=2k$ .

After pass  $2k$ ,  $BC$  has been replaced by  $BD$ , where  $D$  (like  $C$ ) contains  $2^k$  sorted blocks, each of size  $2^k$ . However, we can now complete our stable sort of  $L$  by stably sorting  $B$ , sorting  $D$  with a stable *BLOCKSORT* and stably merging  $B$  with  $D$ . See Figure 4. Thus the entire strategy runs in  $O(n \log n)$  time and  $O(1)$  extra space.

$\underline{E_1 D_1 G_1 B_1} \quad \underline{D_3 E_2 E_3 E_4} \quad \underline{F_1 G_2 H_1 H_2} \quad \underline{C_1 C_2 C_3 D_2} \quad \underline{A_1 A_2 A_3 B_2}$   
 $\underbrace{\hspace{1.5cm}}_B \quad \underbrace{\hspace{1.5cm}}_D$

a) Example list after final merging pass.

$\underline{B_1 D_1 E_1 G_1} \quad \underline{D_3 E_2 E_3 E_4} \quad \underline{F_1 G_2 H_1 H_2} \quad \underline{C_1 C_2 C_3 D_2} \quad \underline{A_1 A_2 A_3 B_2}$   
 $\underbrace{\hspace{1.5cm}}_B \quad \underbrace{\hspace{1.5cm}}_D$

b)  $B$  is sorted.

$\underline{B_1 D_1 E_1 G_1} \quad \underline{A_1 A_2 A_3 B_2 C_1 C_2 C_3 D_2 D_3 E_2 E_3 E_4 F_1 G_2 H_1 H_2}$   
 $\underbrace{\hspace{1.5cm}}_B \quad \underbrace{\hspace{1.5cm}}_D$

c)  $D$  is sorted by blocks.

$A_1 \quad A_2 \quad A_3 \quad B_1 \quad B_2 \quad C_1 \quad C_2 \quad C_3 \quad D_1 \quad D_2 \quad D_3 \quad E_1 \quad E_2 \quad E_3 \quad E_4 \quad F_1 \quad G_1 \quad G_2 \quad H_1 \quad H_2$

d)  $B$  and  $D$  are merged.

Figure 4. Completing the sort.

We observe that major factors that make this scheme so much faster than a direct merge-sort implementation are these: (1) we extract the internal buffer only once, not at every merge operation, (2) we use the buffer in a novel and very efficient fashion for passes 1 through  $k+1$  as we break it into advantageously-sized pieces and pass them across the file, and (3) we avoid unnecessary record movement by delaying the use of *BLOCKSORT* until the final step.

### 5.3 Constant of Proportionality Bounds

As in Section 4.5, we focus on key comparisons and record exchanges, concentrating on the constant of proportionality for the leading (this time,  $O(n \log n)$ ) time complexity term.

Filling the buffer requires no more than  $n \log_2 \sqrt{n} = 0.5n \log_2 n$  comparisons and only a linear number of exchanges. (When  $n$  is not of the special form, this is the only step whose time complexity may increase, since a bigger buffer is used. Even so, the buffer's size is no more than doubled, thereby affecting at most the linear term.) The first merging pass needs at most  $n/2$  comparisons and  $n$  exchanges. In general, pass  $i$ ,  $1 \leq i \leq k+1$ , needs at most  $(n - n/2^i)$  comparisons and  $n$  exchanges. The last merging pass, pass  $2k$ , needs at most  $n + n/2 + O(\sqrt{n})$  comparisons ( $n$  to merge,  $n/2 + O(\sqrt{n})$  to search for the correct blocks) and  $n$  exchanges. In general, pass  $2k+1-j$ ,  $1 \leq j \leq k-1$ , needs at most  $(n + n/2^j) + O(\sqrt{n})$  comparisons and  $n$  exchanges. Therefore, we can balance these leading terms, deriving a cost for the merging passes bounded above by  $2kn < n \log_2 n$  comparisons and  $n \log_2 n$  exchanges. Linear time suffices for the final merging and sorting steps. We conclude that we are guaranteed a worst-case key-comparison and record-exchange grand total not greater than  $2.5n \log_2 n$ .

This worst-case total compares favourably with average-case key-comparison and record-exchange totals for popular *unstable* methods: quick-sort's average-case

figure is a little more than  $1.4n \log_2 n$ ; heap-sort's is about  $2.3n \log_2 n$ . (These values are derived from the analysis in Ref. 14, where we count a single record movement at one third the cost of a two-record exchange.)

### 6. DIRECTIONS FOR CONTINUED RESEARCH

We have presented relatively straightforward and efficient stable merging and sorting strategies that simultaneously optimize both time and space (to within a constant factor). The upper bounds we have derived on constants of proportionality are probably overly pessimistic, representing extreme and possibly unrealizable cases hardly representative of expected behaviour. On the other hand, we again remind the reader that we have brushed aside lower-order terms that can be significant in practice, especially for small files.

Given the obvious importance of merging and sorting, a next logical step along this general line of investigation might be a thorough testing of careful implementations of these algorithms. A great number of factors would likely be relevant to such an empirical study, including the frequency distribution of keys, the percentage of duplicate keys present, the initial sortedness of files, the break-even point at which less sophisticated schemes for small subfiles are used, record length, computer architecture and so on.

Optimistically, we observe that even simpler, more effective optimal time and space strategies are very possible. Also, the design and analysis of time-space optimal *parallel* algorithms is a subject of obvious importance for future study.<sup>8</sup> As block rearrangement strategies become more widely known, we hope that their practical potential for merging, sorting, duplicate-key extraction and related file-processing operations will begin to be better understood.

### REFERENCES

1. S. Carlsson, Splitmerge - A fast stable merging algorithm, *Information Processing Letters* 22 189-192 (1986).
2. E. G. Coffman, Jr. and P. J. Denning, *Operating Systems Theory*, Prentice-Hall, Englewood Cliffs, NJ (1973).
3. K. Dudzinski and A. Dydek, On a stable minimum storage merging algorithm, *Information Processing Letters* 12 5-8 (1981).
4. S. Dvorak and B. Durian, Towards an efficient merging, *Lecture Notes in Computer Science* 233 290-298 (1986).
5. S. Dvorak and B. Durian, Merging by decomposition revisited, *The Computer Journal* 31 553-556 (1988).
6. S. Dvorak and B. Durian, Stable linear time sublinear space merging, *The Computer Journal* 30 372-375 (1987).
7. R. B. K. Dewar, A stable minimum storage algorithm, *Information Processing Letters* 2 162-164 (1974).
8. X. Guan and M. A. Langston, Time-space optimal parallel merging and sorting, *IEEE Transactions on Computers* 40 596-602 (1991).
9. B.-C. Huang and M. A. Langston, Practical in-place merging, *Communications of the ACM* 31 348-352 (1988).
10. B.-C. Huang and M. A. Langston, Stable duplicate-key extraction with optimal time and space bounds, *Acta Informatica* 26 473-484 (1989).
11. B.-C. Huang and M. A. Langston, Stable set and multiset operations in optimal time and space, *Proceedings 7th ACM Symposium on Principles of Database Systems* 288-293 (1988).
12. E. C. Horvath, Stable sorting in asymptotically optimal time and extra space, *Journal of the ACM* 25 177-199 (1978).
13. D. E. H. Jones, *The Inventions of Daedalus*, W. H. Freeman and Co., New York, NY, (1982).
14. D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, MA, (1973).
15. D. E. Knuth, private communication.
16. M. A. Kronrod, An optimal ordering algorithm without a field of operation, *Dok. Akad. Nauk SSSR* 186 1256-1258 (1969).
17. D. Motzin, A stable quicksort, *Software - Practice and Experience* 11 607-611 (1981).
18. H. Mannila and E. Ukkonen, A simple linear-time algorithm for in situ merging, *Information Processing Letters* 18 203-208 (1984).
19. F. P. Preparata, A fast stable sorting algorithm with absolutely minimum storage, *Theoretical Computer Science* 1 185-190 (1975).
20. R. Rivest, A fast stable minimum storage sorting al-

- gorithm, Rep. 43, Institute de Recherche d'Informatique, Rocquencourt, France, (1973).
21. J. S. Salowe and W. L. Steiger, Simplified stable merging tasks, *Journal of Algorithms* 8 557-571 (1987).
  22. L. Trabb Pardo, Stable sorting and merging with optimal space and time bounds, Computer Science Department Technical Report CS-74-470, Stanford University, (1974).
  23. L. Trabb Pardo, Stable sorting and merging with optimal space and time bounds, *SIAM Journal on Computing* 6 351-372 (1977).
  24. L. Trabb Pardo, private communication.
  25. J. K. Wong, Some simple in-place merging algorithms, *BIT* 21 157-166 (1981).

## Book Reviews

ANDREW HARTER. *Three-dimensional Integrated Circuit Layout*. Cambridge University Press. £25. ISBN 0 521 41630 2.

The commercial use of three-dimensional integrated circuit (IC) technologies is not yet with us despite the fact that suitable device structures are beginning to emerge. The organisation of a three-dimensional IC involves the use of vertical layers of devices separated by insulation planes, and can be seen as a development of the use of the silicon-on-insulate structures already encountered in conventional two-dimensional technologies. Potential benefits include higher packing density and speed. This book is *not* concerned with the development of new technologies of this class, but rather seeks to address the question of how to develop an IC layout strategy that fully exploits the potential of the structure, given that it is available.

The book represents the author's doctoral thesis and is one of a few published annually on the basis of selection by a review panel set up jointly by the Conference of Professors of Computer Science and the British Computer Society. The panel's aim is to select for wider dissemination British PhD research theses of outstanding calibre, both in content and technical significance. The author presents, as might be expected given the book's pedigree, a most readable and informative view of the emergence of the three-dimensional technologies, setting this account into the context of conventional two-dimensional processes. The presentation of the many positive benefits of these technologies is balanced by a thorough analysis of their drawbacks and fabrication problems, including yield, heat dissipation and electrical parasitic effects. The industry has shown an extraordinary ability over three decades to overcome problems of exactly this type, and it seems entirely reasonable to suppose that these limitations will be dealt with effectively in time. Harter establishes the additional complexity and richness of connectivity available to the user and then considers the development of appropriate layout methods. One of the results of his investigation is a very comprehensive development and evaluation of a novel abutment-based layout scheme, which can be viewed as a generalisation of the two-dimensional abutment system commonly used in silicon compilers and cell-based automatic or semi-automatic design environments. Issues such as vertical scaling and the optimum number of layers that should be used are addressed. If, indeed, the taxing processing problems intro-

duced are eventually overcome, and if the ever-growing need for on-chip capacity is not met more effectively by one of the other novel process developments, this work will form a most important starting point for the practical use of three-dimensional device technologies.

As a research monograph the book is not, of course, an undergraduate text but can be recommended wholeheartedly to the researcher and practising engineer working in the field. The style adopted by the author also makes the book accessible to the manager who is looking for a comprehensive overview of this topic, which promises much for future generations of advanced, high-density ICs. Very full referencing is provided for the reader who wants to take any particular topic further. Finally, as a model of how to write a PhD thesis, the book is exemplary; at once deep, thorough and far-sighted, the author does not fail to address the downsides as well as the upsides of his topic and presents reflections that are all the more substantial and valuable as a result.

R. E. MASSARA  
University of Essex

F. BRACKX and D. CONSTALES  
*Computer Algebra with LISP and REDUCE*  
Kluwer Academic Dordrecht, The Netherlands  
ISBN 0-7923-1441-7, £54.

In recent years the heavy memory demands of computer algebra (CA) systems have ceased to be a serious handicap. The resulting ready availability of such systems, more than 20 years after their genesis, has at last stimulated authors and publishers to bring out texts on the field. Like another recent book of which I am a co-author, this one deals with Reduce, which is one of the two CA systems, both Lisp-based and still under development, to survive from the 1960s (the other being Macsyma). Reduce, like its competitors, has merits and demerits, and certain unique features: no one system is best for all purposes.

The book is based on lecture courses given at various European universities, and is clearly intended to introduce Reduce to novices. It begins with a brief introduction to CA, defining it and contrasting it with conventional programming languages, and includes a somewhat inaccurate description of other CA languages.

Reduce is written in Rlisp, which is essentially an extension of Standard Lisp written in an ALGOL-like syntax. Reduce itself has

two modes. The algebraic mode, which is the normal one for direct use, parses mathematical expressions and presents a functional programming paradigm. The symbolic mode is Lisp-like, and is used for increased efficiency in programming because it enables more direct access to data structures.

After their introduction, the authors plunge straight into Standard Lisp. The description contains some inaccuracies; more importantly, it looks at Lisp itself from a misleading viewpoint in that it stresses low-level functions and does not adequately deal with Lisp's data abstractions or with the actual ways in which Reduce uses Lisp data structures. As a result this part, 85 pages long, fails to connect with the later chapters; it does not give sufficient information about how Reduce works to enable a programmer to begin to program in symbolic mode.

The main part of the book (95 pages) is a description of algebraic-mode Reduce (which I feel would be a more natural starting point for new users interested in applications). The format is rather like a reference manual, defining, describing and illustrating functions individually, without much guidance on how best to combine them or what difficulties the user might encounter. Curiously, although the book came out late enough to cover the latest Reduce version, 3.4, it does not fully exploit either the new functions such as those for local substitution rulesets or the new packages now distributed.

Finally, the authors give 50 pages of examples, covering a functional iteration, an algebra of projection operators, the Grobner bases package for polynomial ideals, and the Clifford algebra arising from 3-dimensional vectors. Source code is given, except for the Grobner bases and the later parts of the Clifford algebra discussion (the latter being thereby made rather pointless), but little advice or guidance on programming appears, and the reasons for the authors' choices of implementations are not discussed. Nor is their choice of subtitle entirely clarified, since there are many issues of interest to pure mathematicians that are not covered or even mentioned, the range of examples that do appear is restricted, and there is no indication of which areas of mathematics lend themselves to treatment by CA and which do not.

As an author of a rival text, I do not feel very threatened by this competitor, even though it contains some useful remarks and examples. It is also quite expensive.

M. A. H. MACCALLUM  
London

# Parallel Methods for Solving Fundamental File Rearrangement Problems<sup>\*,†</sup>

XIAOJUN GUAN

Engineering Physics and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831

AND

MICHAEL A. LANGSTON

Department of Computer Science, University of Tennessee, Knoxville, Tennessee 37996

We present parallel algorithms for the elementary binary set operations that, given an EREW PRAM with  $k$  processors, operate on two sorted lists of total length  $n$  in  $O(n/k + \log n)$  time and  $O(k)$  extra space and are thus time-space optimal for any value of  $k \leq n/(\log n)$ . Our methods are stable, require no information other than a record's key, and do not modify records as they execute. © 1992 Academic Press, Inc.

## 1. INTRODUCTION

The design and analysis of optimal parallel file rearrangement algorithms have long been topics of widespread attention. The vast majority of the published literature has concentrated on the search for algorithms that are *time optimal*, that is, those that achieve optimal speedup (see, for example, [1]). Unfortunately, space management issues have often taken a back seat in these efforts, leaving those who seek to implement optimal parallel algorithms unable to do so with any reasonable, bounded number of processors.

In recent work, however, parallel merge and sort methods that simultaneously optimize *both* time and space have been devised [2]. Such *time-space optimal* algorithms attain optimal speedup, yet require only a constant amount of extra space per processor, even when the number of processors is fixed. Just what scope of file rearrangement problems is amenable to time-space optimal parallel techniques? In this paper we provide a partial answer to this question, developing time-space optimal parallel algorithms for the elementary binary set

operations, namely, set union, intersection, difference, and exclusive or.

To accomplish our goal, we devise a new parallel *select* procedure, reducing the general problem to one of a series of disjoint local operations, one for each processor, on which we can exploit sequential methods. Given an EREW PRAM with  $k$  processors, our algorithms operate on two sorted lists of total length  $n$  in  $O(n/k + \log n)$  time and  $O(k)$  extra space and are thus time-space optimal for any value of  $k \leq n/(\log n)$ . For the sake of complete generality, our algorithms are stable (records with identical keys retain their original relative order), do not modify records (even temporarily) as they execute, and require no information other than a record's key.

## 2. TIME-SPACE OPTIMAL PARALLEL SELECT ON THE EREW PRAM MODEL

Given two sorted lists  $L1$  and  $L2$ , our goal is to transform  $L1$  into two sorted sublists  $L3$  and  $L4$ , where  $L3$  consists of the records whose keys are not found in  $L2$  and  $L4$  consists of the records whose keys are. Thus we accept  $L = L1L2$  and select records from  $L1$  whose keys are contained in  $L2$ , accumulating them in  $L4$ , where our output is of the form  $L3L4L2$ .

Our parallel algorithm comprises four steps: *local selecting*, *series delimiting*, *blockifying*, and *block rearranging*. To facilitate discussion, we temporarily assume that the number of records of each type ( $L1$ ,  $L2$ ,  $L3$ , and  $L4$ ) is evenly divisible by  $k$ , where  $k$  denotes the number of processors available.

*Local Selecting.* We first view  $L$  as a collection of  $k$  blocks, each of size  $n/k$ , and associate a distinct processor with each block. We seek to treat each  $L1$  block  $L1_i$  as if it were the only block in  $L1$ , transforming its contents into the form  $L3_iL4_i$ .

Our first task in this step is to determine where each tail (rightmost element) of each  $L1$  block would go if the

\* A preliminary version of a portion of this paper was presented at the International Conference on Databases, Parallel Architectures and Applications (PARBASE-90), held in Miami Beach, Florida, in March 1990.

† This research has been supported in part by the National Science Foundation under Grant MIP-8919312 and by the Office of Naval Research under Contract N00014-90-J-1855.

tails alone were to be merged with  $L2$ . In order to make this determination efficiently on the EREW model, we direct each  $L1$  processor to set aside four extra storage cells (for copies of indices, offsets, and keys) and employ the "phased merge" as described in the displacement computing step of the merge in [2]. At most  $O(\log n)$  time and  $O(k)$  extra space have been consumed up to this point.

As long as an  $L1$  processor does not need to consider more than  $O(n/k)$   $L2$  records (a quantity known by considering the difference between where its block's tail would go and where the tail of the block to its immediate left would go if they were to be merged with  $L2$ ), we instruct it to employ the linear-time, in-place sequential select routine from [5]. Otherwise, in the case where an  $L1$  block spans several  $L2$  blocks, we first enlist the aid of the corresponding  $L2$  processors to preprocess their records (performing the time-space optimal sequential select against the  $L1$  block, followed by a time-space optimal sequential duplicate-key extract [4]), then instruct the  $L1$  processor to perform its select (at most  $n/k$   $L2$  records are now needed), and finally direct the  $L2$  processors to restore their blocks (two time-space optimal sequential merge operations suffice).

Thus, if we let  $h$  denote the number of blocks in  $L1$ , the  $L1$  list has now taken on the form  $L3_1 L4_1 L3_2 L4_2 \dots L3_h L4_h$ . This completes the local selecting step and has required  $O(n/k + \log n)$  time and constant extra space per processor.

**Series Delimiting.** We now seek to divide  $L1$  into a collection of nonoverlapping "series," each series with  $n/k$   $L3$  records. To begin this process, we locate special records that we term "breakers," each of which is the  $(m(n/k) + 1)$ th  $L3$  record for some integer  $m$ . First we compute prefix sums on  $|L3_i|$  to find these breakers. For example, if  $\sum_{i=1}^{g-1} |L3_i| < m(n/k) + 1$  and  $\sum_{i=1}^g |L3_i| \geq m(n/k) + 1$ , then block  $g$  contains the  $m$ th breaker. We identify three special types of breakers. If block  $i$  contains a breaker, but neither block  $i - 1$  nor block  $i + 1$  contains breakers, then the breaker in block  $i$  is called a "lone" breaker. If block  $i - 1$  and block  $i$  both contain breakers, and if block  $i + 1$  does not contain a breaker, then the breaker in block  $i$  is called a "trailing" breaker. If block  $i$  and block  $i + 1$  both contain breakers, and block  $i - 1$  does not contain a breaker, then the breaker in block  $i$  is called a "leading" breaker.

These breakers are used to divide  $L1$  into nonoverlapping series as follows: each series begins with a lone or trailing breaker and ends with the record immediately preceding the next lone or leading breaker. By design, each series contains exactly  $n/k$   $L3$  records. A sample series is depicted in Fig. 1, where we use  $L3_f$  to denote  $L3_f$  minus any records that precede its breaker and  $L3_{g+1}$

$$\dots \underbrace{L3_f^- L4_f L3_{f+1} L4_{f+1} \dots L3_{g-1} L4_{g-1} L3_g L4_g L3_{g+1}^-}_{\text{one series}} \dots$$

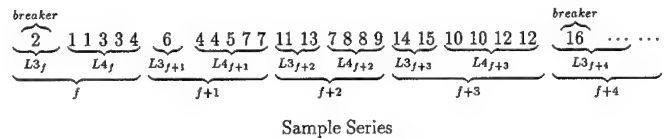
FIG. 1. A sample series obtained in the series delimiting step.

to denote  $L3_{g+1}$  minus its breaker and any records that follow it.

A processor that holds a lone or trailing breaker broadcasts its breaker's location to its right. After that, a processor that holds a lone or leading breaker broadcasts its breaker's location to its left. (This type of broadcasting can be efficiently accomplished on the EREW PRAM with data distribution algorithms or parallel prefix computations.) By this means, a processor learns the location of the lone or trailing breaker to its immediate left and the location of the lone or leading breaker to its immediate right. This completes the series delimiting step and has required  $O(\log(n/k) + \log k)$  time and constant extra space per processor.

**Blockifying.** In this step, we first reorganize in parallel the  $L1$  records within every series and then reorganize in parallel the records in the remainder of the  $L1$  list.

Let us consider our sample series as depicted in Fig. 1. We seek to collect the  $n/k$   $L3$  records in this series in block  $g$  (and thus move the  $L4$  records into the other blocks and subblocks illustrated). It is a simple matter to exchange  $L3_{g+1}^-$  with the rightmost  $|L3_{g+1}^-|$  records in  $L4_g$ . Efficiently coalescing the other  $L3$  records into block  $g$  is much more difficult. We begin by computing prefix sums on  $|L3_f^-|$ ,  $|L3_{f+1}|$ , ...,  $|L3_{g-2}|$ ,  $|L3_{g-1}|$  to obtain a "displacement table." Table entry  $E_i = \sum_{k=f}^i |L3_k|$  denotes the number of  $L3$  records in blocks indexed  $f$  through  $i$  that are to move to block  $g$ . It turns out that  $E_i$  will also denote the number of  $L4$  records that block  $i$  is to receive from block  $i + 1$  as our algorithm proceeds. In Fig. 2, our sample series is shown in more detail (with  $g$



$i$	$E_i$
$f$	1
$f+1$	2
$f+2$	4

Displacement Table

FIG. 2. A more detailed view of a sample series and its displacement table.

set at  $f + 3$ ) along with its corresponding displacement table.

Thus each processor  $i$ ,  $f < i < g$ , now uses the displacement table to determine exactly how the records in its block are to be rearranged: it is to send  $|L3_i|$  records to block  $g$ , send its first  $E_{i-1}$   $L4$  records (which we denote by  $X_i$ ) to block  $i - 1$ , retain its next  $n/k - |L3_i| - E_{i-1}$   $L4$  records (denoted by  $Y_i$ ), and receive  $E_i$   $L4$  records (denoted by  $X_{i+1}$ ) from block  $i + 1$ . Processors  $f$  and  $g$  determine similar information: processor  $f$  is to send  $|L3_f| = E_f$  records to block  $g$  and receive the same number of records from block  $f + 1$ , and processor  $g$  is to send  $|L4_g| = E_{g-1}$  records to block  $g - 1$  and receive the same number of records from blocks  $f$  through  $g - 1$ . (Note that segments  $X_f$  and  $Y_g$  are empty.)

To accomplish the data movement, each processor first reverses the contents of its block and then reverses its  $X$ ,  $Y$ , and  $L3$  segments separately, thereby efficiently permuting its (two or) three subblocks. Each processor  $j$ ,  $f < j \leq g$ , now employs a single extra storage cell to copy safely the first record of  $X_j$  to the location formerly occupied by the first record of  $X_{j-1}$ , while processor  $f$  copies the first record of its  $L3$  segment to the location formerly occupied by the first record of  $X_g$ . Data movement continues in this fashion, with each processor moving its  $L3$  records to block  $g$  as soon as its  $X$  segment is exhausted.

Note that if  $k$  is small enough (no greater than  $O(\max\{n/k, \log n\})$ ), then the displacement table can merely be searched; if  $k$  is larger than this, then the table may contain too many identical entries, and we invoke a preprocessing routine to condense it (again with the aid of broadcasting).

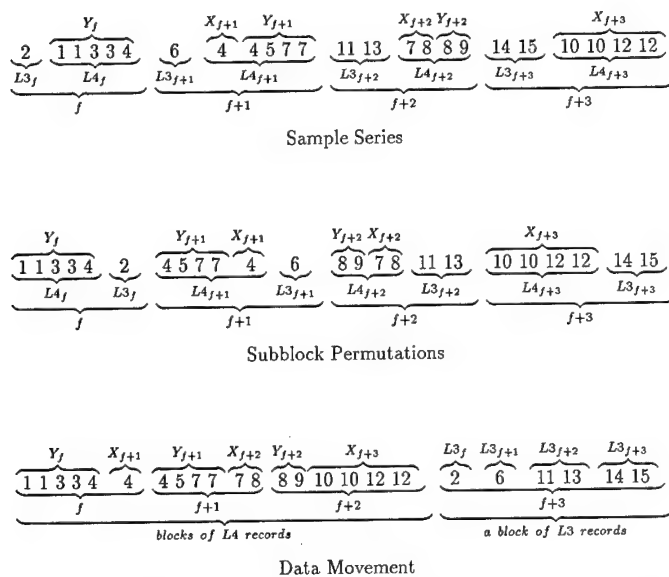


FIG. 3. Coalescing the  $L3$  records of one series into a single block.

After the data movement is finished, it is necessary to rotate  $L3_g$  with the records moved into block  $g$  from block  $g + 1$ . The processing of the series is now completed, as depicted in Fig. 3.

If block  $g + 1$  contains a leading breaker, we next rotate the records in an appropriate prefix of this block to ensure that  $L3$  records precede  $L4$  records there.

We can now handle the records not spanned by a series. These records are contained in zero or more non-overlapping "sequences" (we choose this term to avoid confusion with "series"), where each sequence begins with a leading breaker and ends with the record immediately preceding the next trailing breaker. Suppose such a sequence spans  $p$  blocks. Because there are exactly  $p$  breakers in these blocks and because the  $L3$  records before the first breaker and after the last breaker have been moved outside these blocks, there are now exactly  $(p - 1)(n/k)$   $L3$  records there. Thus, there are exactly  $n/k$   $L4$  records there.

If  $p = 2$ , then the two blocks have the form  $L3_i L4_i L3_{i+1} L4_{i+1}$ , where  $|L4_i| = |L3_{i+1}|$ . Swapping  $L4_i$  with  $L3_{i+1}$  finishes the blockifying for this sequence. If  $p > 2$ , then we simply treat the sequence as we earlier did each series, exchanging the roles of  $L3$  and  $L4$  records. This completes the blockifying step and has required  $O(n/k + \log n)$  time and constant extra space per processor.

**Block Rearranging.**  $L3$  has now become an ordered collection of blocks interspersed with another ordered collection that constitutes  $L4$ . We need only to rearrange these blocks so that  $L3$  is followed by  $L4$ . We direct each processor to set aside a zero bit if it contains an  $L3$  block and to set aside a one bit otherwise. The processors now need only compute prefix sums on these values and then acquire their respective new blocks in parallel without memory conflicts. This completes the block rearranging step and has required  $O(n/k + \log k)$  time and constant extra space per processor.

### 3. IMPLEMENTATION DETAILS

Suppose that the number of  $L3$  (and hence  $L4$ ) records is not evenly divisible by  $k$ , in which case the last breaker begins a series with strictly fewer than  $n/k$   $L3$  records. This series is treated just as any other (although it may be a very short one, lying entirely within the last block). In blockifying, the  $L3$  records in this series are collected in the last block, so that this series becomes a (possibly empty) collection of  $L4$  blocks followed by (possibly just part of) one block containing both  $L3$  and  $L4$  records. After the block rearranging step, we need only move the  $L3$  and  $L4$  segments from the last block into their appropriate final positions with parallel rotations.

More generally, suppose that the number of  $L1$  records is not evenly divisible by  $k$ . (Note that the number of  $L2$  records never really needs to be evenly divisible by  $k$ .) We transform  $L1L2$  into the list  $L1^1L1^2L2$ , where  $L1^1$  contains an integral multiple of  $n/k$  records and where  $L1^2$  contains strictly fewer than  $n/k$  records. We further transform the input into the list  $L1^1L2L1^2$  by means of parallel rotations and invoke the main algorithm on  $L1^1L2$ , yielding  $L3^1L4^1L2L1^2$ . Then a local select of  $L1^2$  against  $L2$  gives  $L3^1L4^1L2L3^2L4^2$ . Parallel rotations now produce the desired result,  $L3^1L3^2L4^1L4^2L2 = L3L4L2$ .

The time and space requirements of these implementation details are thus bounded by those of the main parallel algorithm. This completes the description of our parallel method. In summary, the total time spent is  $O(n/k + \log n)$  and the total extra space used is  $O(k)$ . Therefore, this select algorithm is time-space optimal for any value of  $k \leq n/(\log n)$ , thereby meeting our stated goal.

#### 4. TIME-SPACE OPTIMAL PARALLEL SET OPERATIONS

In what follows, suppose we are given the input list  $L = XY$ , where  $X$  and  $Y$  are two sublists, each sorted on the key and each containing no duplicates. Since the same key may naturally appear once in  $X$  and once in  $Y$ , we insist that, in the spirit of stability, the record represented in the result of a binary set operation be the one that occurs first in  $L$ .

We now have stable, time-space optimal parallel subroutines sufficient to perform the elementary binary set operations. Select is obtained from the work of the last two sections. Merge is obtained from [2]. Duplicate-key extract is obtained from an easy modification to select, in which we replace the first step, local selecting, with the local duplicate-key extracting method of [4]. (Local duplicate-key extract is actually easier than local select, because the  $L1$  processors need no information from the  $L2$  list.)

We invoke merge followed by duplicate-key extract to produce  $X \cup Y$ . We perform select to yield both  $X \cap Y$  and  $X - Y$ . To achieve  $X \oplus Y$ , we invoke select on  $XY$ , producing  $X_1X_2Y$ ; rotate  $X_2$  and  $Y$  to yield  $X_1YX_2$ ; perform select on  $YX_2$ , producing  $X_1Y_1Y_2X_2$ ; and finally merge  $X_1$  and  $Y_1$ .

#### 5. CONCLUDING REMARKS

Assuming only the weak EREW PRAM model, we have presented for the first time parallel algorithms for

the elementary binary set operations that are asymptotically time-space optimal. As a bonus, these methods immediately extend to multisets (under several natural definitions [5]). Although  $n$  must be large enough to satisfy the inequality  $k \leq n/(\log n)$  for optimality, we observe that our algorithms are also *efficient* in the usual sense (their speedup is within a polylogarithmic factor of optimal) for any value of  $n$ , suggesting that they may have practical merit even for relatively small files. As long as main memory remains a critical resource in many environments, the quest for techniques that permit the efficient use of both time and space continues to be a fertile research domain.

#### REFERENCES

1. Akl, S. G., and Santoro, N. Optimal parallel merging and sorting without memory conflicts. *IEEE Trans. Comput.* **36** (1987), 1367-1369.
2. Guan, X., and Langston, M. A. Time-space optimal parallel merging and sorting. *IEEE Trans. Comput.* **40** (1991), 596-602.
3. Huang, B.-C., and Langston, M. A. Practical in-place merging. *Comm. ACM* **31** (1988), 348-352.
4. Huang, B.-C., and Langston, M. A. Stable duplicate-key extraction with optimal time and space bounds. *Acta Inform.* **26** (1989), 473-484.
5. Huang, B.-C., and Langston, M. A. Stable set and multiset operations in optimal time and space. *Proc. 7th ACM Symposium on Principles of Database Systems*, 1988, pp. 288-293.

---

XIAOJUN GUAN received his B.S. and M.S. degrees in computer science from Jilin University, China, in 1982 and 1985, respectively, and his Ph.D. degree in computer science from Washington State University in 1990. He is currently employed as a postdoctoral research associate at Oak Ridge National Laboratory. His current research interests include parallel and distributed processing, database management, and artificial intelligence.

MICHAEL A. LANGSTON was born on April 21, 1950, in Glen Rose, Texas. He received his Ph.D. degree in computer science from Texas A&M University in 1981. From 1981 to 1989 he served on the faculty at Washington State University. Since that time he has been on the faculty at the University of Tennessee. Dr. Langston has authored over 50 refereed journal papers. His research has been supported by the National Science Foundation and the Office of Naval Research. His current research interests include the design and analysis of algorithms, concrete complexity theory, graph theory, operations research, and VLSI.

## ON WELL-PARTIAL-ORDER THEORY AND ITS APPLICATION TO COMBINATORIAL PROBLEMS OF VLSI DESIGN\*

MICHAEL R. FELLOWS<sup>†</sup> AND MICHAEL A. LANGSTON<sup>‡</sup>

**Abstract.** The existence of decision algorithms with low-degree polynomial running times for a number of well-studied graph layout, placement, and routing problems is nonconstructively proved. Some were not previously known to be in  $\mathcal{P}$  at all; others were only known to be in  $\mathcal{P}$  by way of brute force or dynamic programming formulations with unboundedly high-degree polynomial running times. The methods applied include the recent Robertson–Seymour theorems on the well-partial-ordering of graphs under both the minor and immersion orders. The complexity of search versions of these problems is also briefly addressed.

**Key words.** nonconstructive proofs, polynomial-time complexity, well-partially-ordered sets

**AMS(MOS) subject classifications.** 68C25, 68E10, 68K05

**1. Introduction.** Practical problems are often characterized by fixed-parameter instances. In the VLSI domain, for example, the parameter may represent the number of tracks permitted on a chip, the number of processing elements to be employed, the number of channels required to connect circuit elements, or the load on communications links. In fixing the value of such parameters, we help focus on the physically realizable nature of the system rather than on the purely abstract aspects of the model.

In this paper, we employ and extend Robertson–Seymour poset techniques to prove low-degree polynomial-time decision complexity for a variety of fixed-parameter layout, placement, and routing problems, dramatically lowering known time-complexity upper bounds. Our main results are summarized in Table 1, where  $n$  denotes the number of vertices in an input graph and  $k$  denotes the appropriate fixed parameter. (At the referee's urging, we also list relevant, previously published results from [5], [8], as noted in the rightmost column of the table.)

In the next section, we survey the necessary background from graph theory and graph algorithms that makes these advances possible. Sections 3–5 describe our results on several representative types of decision problems, illustrating a range of techniques based on well-partially-ordered sets. In §6, we discuss how self-reducibility can be used to bound the complexity of search versions of these problems. A few open problems and related issues are briefly addressed in the final section.

**2. Background.** Except where explicitly noted otherwise, all graphs that we consider are finite and undirected. A graph  $H$  is less than or equal to a graph  $G$  in the *minor* order, written  $H \leq_m G$ , if and only if a graph isomorphic to  $H$  can be obtained from  $G$  by a series of these two operations: taking a subgraph and contracting an edge. For example, the construction depicted in Fig. 1 shows that  $W_4 \leq_m Q_3$ .

\* Received by the editors January 5, 1990; accepted for publication (in revised form) March 13, 1991. A preliminary version of a portion of this paper was presented at the Fifth MIT Conference on Advanced Research in VLSI held in Cambridge, Massachusetts in March 1988.

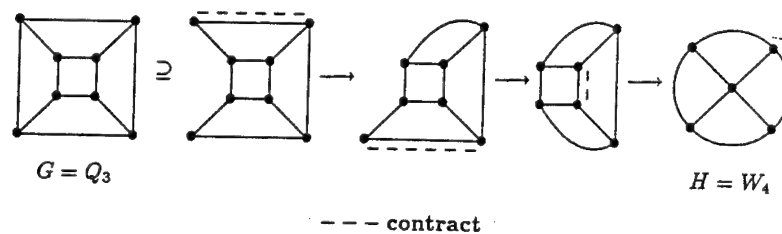
<sup>†</sup> Department of Computer Science, University of Victoria, Victoria, British Columbia, Canada V8W 2Y2. This author's research was supported in part by National Science Foundation grant MIP-8919312 and by Office of Naval Research contract N00014-88-K-0456.

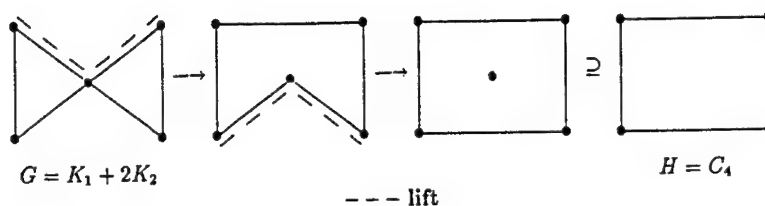
<sup>‡</sup> Department of Computer Science, University of Tennessee, Knoxville, Tennessee 37996-1301. This author's research was supported in part by National Science Foundation grant MIP-8919312 and by Office of Naval Research contract N00014-88-K-0343.

TABLE 1  
Main results.

General problem area	Problem	Best previous upper bound	Our result
circuit layout	GATE MATRIX LAYOUT	open	$O(n^2)$ [5]
Linear arrangement	MIN CUT LINEAR ARRANGEMENT	$O(n^{k-1})$	$O(n^2)$
	MODIFIED MIN CUT	$O(n^k)$	$O(n^2)$
	TOPOLOGICAL BANDWIDTH*	$O(n^k)$	$O(n^2)$ [8]
	VERTEX SEPARATION	$O(n^{k^2+2k+4})$	$O(n^2)$
Circuit design and utilization	CROSSING NUMBER*	open	$O(n^3)$ [8]
	MAX LEAF SPANNING TREE	$O(n^{2k+1})$	$O(n^2)$
	SEARCH NUMBER	$O(n^{2k^2+4k+8})$	$O(n^2)$
Embedding and routing	2-D GRID LOAD FACTOR	open	$O(n^2)$
	BINARY TREE LOAD FACTOR	open	$O(n^2)$
	DISK DIMENSION	open	$O(n^3)$ [5]
	EMULATION	open	$O(n^3)$ [8]

\* Input restricted to graphs of maximum degree three.

FIG. 1. Construction demonstrating that  $W_4$  is a minor of  $Q_3$ .

FIG. 2. Construction demonstrating that  $C_4$  is immersed in  $K_1 + 2K_2$ .

Note that the relation  $\leq_m$  defines a partial ordering on graphs. A family  $F$  of graphs is said to be *closed* under the minor ordering if the facts that  $G$  is in  $F$  and that  $H \leq_m G$  together imply that  $H$  must be in  $F$ . The *obstruction set* for a family  $F$  of graphs is the set of graphs in the complement of  $F$  that are minimal in the minor ordering. Therefore, if  $F$  is closed under the minor ordering, it has the following characterization:  $G$  is in  $F$  if and only if there is no  $H$  in the obstruction set for  $F$  such that  $H \leq_m G$ .

THEOREM 2.1 (see [25]). *Graphs are well-partially-ordered<sup>1</sup> by  $\leq_m$ .*

THEOREM 2.2 (see [24]). *For every fixed graph  $H$ , the problem that takes as input a graph  $G$  and determines whether  $H \leq_m G$  is solvable in polynomial time.*

Theorems 2.1 and 2.2 guarantee the *existence* of a polynomial-time decision algorithm for any minor-closed family of graphs, but do not provide any details of what that algorithm might be. Moreover, no proof of Theorem 2.1 can be entirely constructive. For example, there can be no systematic method of computing the finite obstruction set for an arbitrary minor-closed family  $F$  from the description of a Turing machine that precisely accepts the graphs in  $F$  [9].

An interesting feature of Theorems 2.1 and 2.2 is the low degree of the polynomials bounding the decision algorithms' running times (although the constants of proportionality are enormous). Letting  $n$  denote the number of vertices in  $G$ , the time required to recognize  $F$  is  $O(n^3)$ . If  $F$  excludes a planar graph, then  $F$  has bounded tree-width [22] and the time complexity decreases to  $O(n^2)$ .

A graph  $H$  is less than or equal to a graph  $G$  in the *immersion* order, written  $H \leq_i G$ , if and only if a graph isomorphic to  $H$  can be obtained from  $G$  by a series of these two operations: taking a subgraph and *lifting*<sup>2</sup> a pair of adjacent edges. For example, the construction depicted in Fig. 2 shows that  $C_4 \leq_i K_1 + 2K_2$  (although  $C_4 \not\leq_m K_1 + 2K_2$ ).

The relation  $\leq_i$ , like  $\leq_m$ , defines a partial ordering on graphs with the associated notions of closure and obstruction sets.

THEOREM 2.3 (see [21]). *Graphs are well-partially-ordered by  $\leq_i$ .*

<sup>1</sup> A partially-ordered set  $(X, \leq)$  is *well-partially-ordered* if (1) any subset of  $X$  has finitely many minimal elements and (2)  $X$  contains no infinite descending chain  $x_1 \geq x_2 \geq x_3 \geq \dots$  of distinct elements.

<sup>2</sup> A pair of adjacent edges  $uv$  and  $vw$ , with  $u \neq v \neq w$ , is *lifted* by deleting the edges  $uv$  and  $vw$  and adding the edge  $uw$ .

The proof of the following result is original, although it has been independently observed by others as well [20].

**THEOREM 2.4.** *For every fixed graph  $H$ , the problem that takes as input a graph  $G$  and determines whether  $H \leq_i G$  is solvable in polynomial time.*

*Proof.* Letting  $k$  denote the number of edges in  $H$ , we replace  $G = \langle V, E \rangle$  with  $G' = \langle V', E' \rangle$ , where  $|V'| = k|V| + |E|$  and  $|E'| = 2k|E|$ . Each vertex in  $V$  is replaced in  $G'$  with  $k$  vertices. Each edge  $e$  in  $E$  is replaced in  $G'$  with a vertex and  $2k$  edges connecting this vertex to all of the vertices that replace  $e$ 's endpoints. We can now apply the disjoint-connecting paths algorithm of [24], since it follows that  $H \leq_i G$  if and only if there exists an injection from the vertices of  $H$  to the vertices of  $G'$  such that each vertex of  $H$  is mapped to some vertex in  $G'$  that replaces a distinct vertex from  $G$ , and such that  $G'$  contains a set of  $k$  vertex-disjoint paths, each one connecting the images of the endpoints of a distinct edge in  $H$ .  $\square$

Theorems 2.3 and 2.4, like Theorems 2.1 and 2.2, only guarantee the existence of a polynomial-time decision algorithm for any immersion-closed family  $F$  of graphs. The method we use in proving Theorem 2.4 yields an obvious time bound of  $O(n^{h+6})$ , where  $h$  denotes the order of the largest graph in  $F$ 's obstruction set. (There are  $O(n^h)$  different injections to consider; the disjoint-paths algorithm takes cubic time on  $G'$ , a graph of order at most  $n^2$ .) Thanks to the next theorem of Mader, however, we find that the bound immediately reduces to  $O(n^{h+3})$  because the problem graphs of interest permit only a linear number of distinct edges.

**THEOREM 2.5** (see [14]). *For any graph  $H$  there exists a constant  $c_H$  such that every simple graph  $G = \langle V, E \rangle$  with  $|E| > c_H|V|$  satisfies  $G \geq_i H$ .*

We show in §4 that by exploiting excluded-minor knowledge on immersion-closed families the time complexity for determining membership can, in many cases, be reduced to  $O(n^2)$ .

**3. Exploiting the minor order.** Given a graph  $G$  of order  $n$ , a linear layout of  $G$  is a bijection  $\ell$  from  $V$  to  $\{1, 2, \dots, n\}$ . For such a layout  $\ell$ , the vertex separation at location  $i$ ,  $s_\ell(i)$ , is  $|\{u : u \in V, \ell(u) \leq i, \text{ and there is some } v \in V \text{ such that } uv \in E \text{ and } \ell(v) > i\}|$ . The vertex separation of the entire layout is  $s_\ell = \max\{s_\ell(i) : 1 \leq i \leq n\}$ , and the vertex separation of  $G$  is  $vs(G) = \min\{s_\ell : \ell \text{ is a linear layout of } G\}$ .

Given both  $G$  and a positive integer  $k$ , the  $\mathcal{NP}$ -complete VERTEX SEPARATION problem [13] asks whether  $vs(G)$  is less than or equal to  $k$ . It has previously been claimed that VERTEX SEPARATION can be decided in  $O(n^{k^2+2k+4})$  time [4], and is thus in  $\mathcal{P}$  for any fixed value of  $k$ . We now prove that the problem can be solved in time bounded by a polynomial in  $n$ , the degree of which does not depend on  $k$ .

**THEOREM 3.1.** *For any fixed  $k$ , VERTEX SEPARATION can be decided in  $O(n^2)$  time.*

*Proof.* Let  $k$  denote any fixed positive integer. We show that the family  $F$  of "yes" instances is closed under the minor ordering. To do this, we must prove that if  $vs(G) \leq k$  then  $vs(H) \leq k$  for every  $H \leq_m G$ . Without loss of generality, we assume that  $H$  is obtained from  $G$  by exactly one of these three actions: deleting an edge, deleting an isolated vertex, or contracting an edge.

If  $H$  is obtained from  $G$  by deleting an edge, then  $vs(H) \leq vs(G) \leq k$  because the vertex separation of any layout of  $G$  either remains the same or decreases by 1 with the removal of an edge. If  $H$  is obtained from  $G$  by deleting an isolated vertex, then, also clearly,  $vs(H) \leq k$ .

Suppose that  $H$  is obtained from  $G$  by contracting the edge  $uv$ . Let  $\ell$  denote a

layout of  $G$  whose vertex separation does not exceed  $k$  and assume that  $\ell(u) < \ell(v)$ . We contract  $uv$  to  $u$  in the layout  $\ell'$  of  $H$  as follows: we set  $\ell'(x) = \ell(x)$  if  $\ell(x) < \ell(v)$  and set  $\ell'(x) = \ell(x) - 1$  if  $\ell(x) > \ell(v)$ . Let us consider the effect of this action on the vertex separation at each location of the layout. Clearly,  $s_{\ell'}(i) = s_{\ell}(i)$  for  $1 \leq i < \ell(u)$ . If there exists a vertex  $w$  with  $\ell(w) > \ell(u)$  and either  $uw \in E$  or  $vw \in E$ , then  $s_{\ell'}(\ell(u)) \leq s_{\ell}(\ell(u)) - 1$ . Similar arguments establish that  $s_{\ell'}(i) \leq s_{\ell}(i)$  for the ranges  $\ell(u) < i < \ell(v)$  and  $\ell(v) \leq i < n$ . Therefore, the vertex separation of  $\ell'$  does not exceed  $k$  and  $vs(H) \leq k$ .

We conclude that, in any case,  $H$  is in  $F$ , and hence  $F$  is minor-closed. It remains only to note that there are trees with arbitrarily large vertex separation (such an excluded planar graph ensures bounded tree-width, and thus a time complexity of  $O(n^2)$ ).  $\square$

Given a graph  $G$  and a positive integer  $k$ , the  $\mathcal{NP}$ -complete SEARCH NUMBER problem [19] asks whether  $k$  searchers are sufficient to ensure the capture of a fugitive who is free to move with arbitrary speed about the edges of  $G$ , with complete knowledge of the location of the searchers. More precisely, we say that every edge of  $G$  is initially *contaminated*. An edge  $e = uv$  becomes *clear* either when a searcher is moved from  $u$  to  $v$  ( $v$  to  $u$ ) while another searcher remains at  $u$  ( $v$ ), or when all edges incident on  $u$  ( $v$ ) except  $e$  are clear and a searcher at  $u$  ( $v$ ) is moved to  $v$  ( $u$ ). (A clear edge  $e$  becomes recontaminated if the movement of a searcher produces a path without searchers between a contaminated edge and  $e$ .) The goal is to determine if there exists a sequence of *search steps* that results in all edges being clear simultaneously, where each such step is one of the following three operations: (1) place a searcher on a vertex, (2) move a searcher along an edge, or (3) remove a searcher from a vertex. It has been reported that SEARCH NUMBER is decidable in  $O(n^{2k^2+4k+8})$  time [4]. As has been independently noted by Papadimitriou [18], however, minor-closure can be applied to reduce this bound.

**THEOREM 3.2.** *For any fixed  $k$ , SEARCH NUMBER can be decided in  $O(n^2)$  time.*

*Proof.* The proof is straightforward by showing that, for fixed  $k$ , the family of "yes" instances is closed under the minor ordering and by observing that there are excluded trees.  $\square$

Consider next the  $\mathcal{NP}$ -complete MAX LEAF SPANNING TREE problem [11]. Given a connected graph  $G$  and a positive integer  $k$ , this problem asks whether  $G$  possesses a spanning tree in which  $k$  or more vertices have degree one. This problem can be solved by brute force in  $O(n^{2k+1})$  time. (There are  $\binom{n}{k}$  ways to select  $k$  leaves and  $O(n)$  possible adjacencies to consider at each leaf. For each of these  $O(n^{2k})$  candidate solutions, the connectivity of the remainder of  $G$  can be determined in linear time because there can be at most a linear number of edges.) Although this means that MAX LEAF SPANNING TREE is in  $\mathcal{P}$  for any fixed  $k$ , we seek to exploit minor-closure so as to ensure a low-degree polynomial running time.

**THEOREM 3.3.** *For any fixed  $k$ , MAX LEAF SPANNING TREE can be decided in  $O(n^2)$  time.*

*Proof.* Let  $k$  denote any fixed positive integer. Consider the proper subset of the "no" instances, the family  $F$  of graphs, none of whose connected components has a spanning tree with  $k$  or more leaves.  $F$  is clearly closed under the minor ordering, from which the theorem follows because we need only test an input graph for connectedness and nonmembership in  $F$ .  $\square$

**4. Exploiting the immersion order.** An *embedding* of an arbitrary graph  $G$  into a fixed *constraint graph*  $C$  is an injection  $f: V(G) \rightarrow V(C)$  together with an assignment, to each edge  $uv$  of  $G$ , of a path from  $f(u)$  to  $f(v)$  in  $C$ . The *minimum load factor* of  $G$  relative to  $C$  is the minimum, over all embeddings of  $G$  in  $C$ , of the maximum number of paths in the embedding that share a common edge in  $C$ .

For example, for the case in which  $C$  is the infinite-length one-dimensional grid, the minimum load factor of  $G$  with respect to  $C$  is called the *cutwidth* of  $G$ . In the  $\mathcal{NP}$ -complete MIN CUT LINEAR ARRANGEMENT problem [11], we are given a graph  $G$  and an integer  $k$ , and are asked whether the cutwidth of  $G$  is no more than  $k$ . Related  $\mathcal{NP}$ -complete problems address the cutwidth of  $G$  relative to  $C$  when  $C$  is the infinite-length, fixed-width two-dimensional grid (2-D GRID LOAD FACTOR) or when  $C$  is the infinite-height binary tree (BINARY TREE LOAD FACTOR).

**THEOREM 4.1.** *For any fixed  $k$  and any fixed  $C$ , the family of graphs for which the minimum load factor relative to  $C$  is less than or equal to  $k$  is closed under the immersion ordering.*

*Proof.* Let an embedding  $f$  of  $G$  in  $C$  with load factor no more than  $k$  be given. Suppose that  $H \leq_i G$ . If  $H \subseteq G$ , then the embedding that restricts  $f$  to  $H$  clearly has load factor no more than  $k$ . If  $H$  is obtained from  $G$  by lifting the edges  $uv$  and  $vw$  incident at vertex  $v$ , then an embedding for  $H$  can be defined by assigning to the resulting edge  $uw$  the composition of the paths from  $u$  to  $v$  and from  $v$  to  $w$  in  $C$ . This cannot increase the load factor.  $\square$

**COROLLARY 4.2.** *For any fixed  $k$ , MIN CUT LINEAR ARRANGEMENT, 2-D GRID LOAD FACTOR, and BINARY TREE LOAD FACTOR can be decided in polynomial time.*

This result has previously been reported for MIN CUT LINEAR ARRANGEMENT, using an algorithm with time complexity  $O(n^{k-1})$  [16]. We now prove that it is sometimes possible to employ excluded-minor knowledge on immersion-closed families to guarantee quadratic-time decision complexity.

**THEOREM 4.3.** *For any fixed  $k$ , MIN CUT LINEAR ARRANGEMENT, 2-D GRID LOAD FACTOR, and BINARY TREE LOAD FACTOR can be decided in  $O(n^2)$  time.*

*Proof.* For MIN CUT LINEAR ARRANGEMENT, it is known that there are binary trees with cutwidth exceeding  $k$  for any fixed  $k$  [2]. Let  $T$  denote such a tree. Because  $T$  has maximum degree three, it follows that  $G \geq_m T$  implies  $G \geq_i T$ . Thus no  $G \geq_m T$  can be a "yes" instance (recall that the "yes" family is immersion closed) and we know from [22] that all "yes" instances have bounded tree-width. (Tree-width and the associated metric branch-width are defined and related to each other in [23].) Now we need only search for a satisfactory tree-decomposition, using the  $O(n^2)$  method of [24]. Testing for obstruction containment in the immersion order can be done in linear time on graphs of bounded tree-width in this setting [24], given such a tree-decomposition.

Sufficiently large binary trees are excluded for 2-D GRID LOAD FACTOR as well (recall that both  $k$  and the grid-width are fixed).

For BINARY TREE LOAD FACTOR, it is a simple exercise to see that all "yes" instances have bounded tree-width by building a tree-decomposition with width at most  $3k$  from a binary tree embedding with load factor at most  $k$ . (The decomposition tree  $T$  can be taken to be the finite subtree of  $C$  that spans the image of  $G$ . For vertex  $u \in V(T)$ , the associated set of vertices of  $G$  contains the inverse image of  $u$  if one exists, and every vertex  $v \in V(G)$  with an incident edge that is assigned a path in  $C$

that includes  $u$ .)  $\square$

**5. Other methods.** The application of Theorems 2.1–2.4 directly ensures polynomial-time decidability. A less direct approach relies on the well-known notion of polynomial-time transformation, as we now illustrate with an example. The  $\mathcal{NP}$ -complete MODIFIED MIN CUT problem was first introduced in [13]. Given a linear layout  $\ell$  of a simple graph  $G$ , the *modified cutwidth* at location  $i$ ,  $c_\ell(i)$ , is  $|\{e : e = uv \in E \text{ such that } \ell(u) < i \text{ and } \ell(v) > i\}|$ . The modified cutwidth of the entire layout is  $c_\ell = \max\{c_\ell(i) : 1 \leq i \leq n\}$ , and the modified cutwidth of  $G$  is  $mc(G) = \min\{c_\ell : \ell \text{ is a linear layout of } G\}$ . Given both  $G$  and a positive integer  $k$ , the MODIFIED MIN CUT problem asks whether  $mc(G)$  is less than or equal to  $k$ . Observe that, while the MIN CUT LINEAR ARRANGEMENT problem addresses the number of edges that cross any cut *between* adjacent vertices in a linear layout, the MODIFIED MIN CUT problem addresses the number of edges that cross (and do not end at) any cut *on* a vertex in the layout.

When  $k$  is fixed, neither the family of “yes” instances nor the family of “no” instances for MODIFIED MIN CUT is closed under either of the available orders. Nevertheless, we can employ a useful consequence of well-partially-ordered sets.

**CONSEQUENCE** (see [8]). *If  $(S, \leq)$  is a well-partially-ordered set that supports polynomial-time order tests for every fixed element of  $S$ , and if there is a polynomial-time computable map  $t: D \rightarrow S$  such that for  $F \subseteq D$ , (a)  $t(F) \subseteq S$  is closed under  $\leq$  and (b)  $t(F) \cap t(D - F) = \emptyset$ , then there is a polynomial-time decision algorithm to determine for input  $z$  in  $D$  whether  $z$  is in  $F$ .*

To use this result on fixed- $k$  MODIFIED MIN CUT, observe that if any vertex of a simple graph  $G$  has degree greater than  $2k + 2$ , then  $G$  is automatically a “no” instance. Given a simple graph  $G$  with maximum degree less than or equal to  $2k + 2$ , we first augment  $G$  with loops as follows: if a vertex  $v$  has degree  $d < 2k + 2$ , then it receives  $(2k + 2) - d$  new loops. Letting  $G'$  denote this augmented version of  $G$ , we now replace  $G'$  with the Boolean matrix  $M$ , in which each row of  $M$  corresponds to an edge of  $G'$  and each column of  $M$  corresponds to a vertex of  $G'$ . That is,  $M$  has  $|E'|$  rows and  $n$  columns, with  $M_{ij} = 1$  if and only if edge  $i$  is incident on vertex  $j$ .  $M$  and  $k' = 3k + 2$  are now viewed as input to the GATE MATRIX LAYOUT problem [3], in which we are asked whether the columns of  $M$  can be permuted so that, if in each row we change to \* every 0 lying between the row's leftmost and rightmost 1, then no column contains more than  $k$  1s and \*s. Thus a permutation of the columns of  $M$  corresponds to a linear layout of  $G$ . For such a permutation, each \* in column  $i$ ,  $1 < i < n$ , represents a distinct edge crossing a cut at vertex  $i$  in the corresponding layout of  $G$ .

**THEOREM 5.1.** *For any fixed  $k$ , MODIFIED MIN CUT can be decided in  $O(n^2)$  time.*

*Proof.* We apply the consequence, using the set of all graphs for  $S$ ,  $\leq_m$  for  $\leq$ , the set of simple graphs of maximum degree  $2k + 2$  for  $D$ , the family of “yes” instances in  $D$  for  $F$ , and the composition of the map just defined from graphs to matrices with the map of [5] from matrices to graphs for  $t$ . Testing for membership in  $D$  and computing  $t$  are easily accomplished in  $O(n^2)$  time. That  $t(F)$  is closed under  $\leq_m$  and excludes a planar graph for any fixed  $k$  is established in [5]. Finally, condition (b) holds because, for any  $G$  in  $D$ ,  $t(G)$  is a “yes” instance for GATE MATRIX LAYOUT with parameter  $3k + 2$  if and only if  $G$  is a “yes” instance for MODIFIED MIN CUT with parameter  $k$ .  $\square$

**6. Search problems.** Given a decision problem  $\Pi_D$  and its search version  $\Pi_S$ , any method that pinpoints a solution to  $\Pi_S$  by repeated calls to an algorithm that answers  $\Pi_D$  is termed a *self-reduction*. This simple notion has been formalized with various refinements in the literature, but the goal remains the same: to use the existence of a decision algorithm to prove the existence of a search algorithm. Note the crucial importance of self-reducibility in the current context, given that Theorems 2.1–2.4 only yield decision algorithms, not search procedures.

It sometimes suffices to *fatten up* a graph by adding edges to isolate a solution. For example, this strategy can be employed to construct solutions to (fixed- $k$ ) GATE MATRIX LAYOUT, when any exist, in  $O(n^4)$  time [1]. It follows from the proof of Theorem 5.1 that the same can be said for MODIFIED MIN CUT as well. We leave it to the reader to verify that such a scheme works for the search version of (fixed- $k$ ) VERTEX SEPARATION, by attempting to add each edge in  $V \times V - E$  in arbitrary order, retaining in turn only those whose addition maintains a “yes” instance, and at the end reading off a satisfactory layout (from right to left) by successively removing a vertex of smallest degree. This self-reduction automatically solves the search version of SEARCH NUMBER, also (see the discussion of “2-expansions” in [4]).

Conversely, it is sometimes possible to *trim down* a graph by deleting edges so as to isolate a solution. It is easy to see that this simple strategy yields an  $O(n^4)$ -time algorithm for the search version of (fixed- $k$ ) MAX LEAF SPANNING TREE, by attempting to delete each edge in  $E$  in arbitrary order, retaining in turn only those whose deletion does not maintain a “yes” instance.

Another technique involves the use of graph *gadgets*. A simple gadget, consisting of two new vertices with  $k$  edges between them, is useful in constructing a solution to (fixed- $k$ ) MIN CUT LINEAR ARRANGEMENT, when any exist, in  $O(n^4)$  time [1]. A similar use of gadgets enables efficient self-reductions for load factor problems. (On BINARY TREE LOAD FACTOR, for example, we can begin by using two  $k$ -edge gadgets  $uv$  and  $wx$  to locate a vertex  $y$  of the input graph that can be mapped to a leaf of the constraint tree by identifying  $u$ ,  $w$ , and  $y$ .)

Indeed, polynomial-time self-reductions exist for all of the problems that we study in this paper. In addition to the straightforward methods just mentioned, faster but more elaborate techniques are described in [6], [9].

**7. Concluding remarks.** The range of problems amenable to an approach based on well-partially-ordered sets is remarkable. Although the problems that we have addressed in this paper are all fixed-parameter versions of problems that are  $\mathcal{NP}$ -hard in general, we remind the reader that by fixing parameters we do not automatically trivialize problems, and thereby obtain polynomial-time decidability (consider, for example, GRAPH  $k$ -COLORABILITY [11]). Moreover, the techniques that we have employed can be used to guarantee membership in  $\mathcal{P}$  for problems that have no associated (fixed) parameter [8].

Table 1 suffers from one notable omission, namely, BANDWIDTH [11]. The only success reported to date has concerned restricted instances of TOPOLOGICAL BANDWIDTH. Both BANDWIDTH and the related EDGE BANDWIDTH problem [7] have resisted this general line of attack so far. Clearly, BANDWIDTH is at least superficially similar to other layout permutation problems we have addressed, and fixed- $k$  BANDWIDTH, like the others, is solvable in (high-degree) polynomial-time with dynamic programming [12]. Perhaps BANDWIDTH, however, is really different; it is one of the very few problems that remain  $\mathcal{NP}$ -complete when restricted to trees [10].

The results that we have derived here immediately extend to hypergraph problem variants as long as hypergraph instances can be efficiently reduced to graph instances. For example, such reductions are known for HYPERGRAPH VERTEX SEPARATION and HYPERGRAPH MODIFIED MIN CUT [17], [27].

Finally, we observe that even partial-orders that fail to be well-partial-orders (on the set of all graphs) may be useful. For example, although it is well known that graphs are not well-partially-ordered under the topological order, it has been shown [15] that, for every fixed  $h$ , all graphs without  $h$  vertex-disjoint cycles are well-partially-ordered under topological containment. Also, polynomial-time order tests exist [24]. Problems such as (fixed- $k$ ) TOPOLOGICAL BANDWIDTH, therefore, are decidable in polynomial time as long as the input is restricted to graphs with no more than  $h$  disjoint cycles (for fixed  $h$ ). Similarly, we might employ the result [26] that graphs without a path of length  $h$ , for  $h$  fixed, are well-partially-ordered under subgraph containment.

**Acknowledgment.** We wish to express our gratitude to the anonymous referee whose extremely careful review of the original version of this paper greatly helped to improve and streamline the final presentation.

## REFERENCES

- [1] D. J. BROWN, M. R. FELLOWS, AND M. A. LANGSTON, *Polynomial-time self-reducibility: Theoretical motivations and practical results*, Internat. J. Comput. Math., 31 (1989), pp. 1-9.
- [2] M.-J. CHUNG, F. MAKEDON, I. H. SUDBOROUGH, AND J. S. TURNER, *Polynomial time algorithms for the min cut problem on degree restricted trees*, SIAM J. Comput., 14 (1985), pp. 158-177.
- [3] N. DEO, M. S. KRISHNAMOORTHY, AND M. A. LANGSTON, *Exact and approximate solutions for the gate matrix layout problem*, IEEE Trans. Computer-Aided Design, 6 (1987), pp. 79-84.
- [4] J. A. ELLIS, I. H. SUDBOROUGH, AND J. S. TURNER, *Graph separation and search number*, in Proc. 21st Allerton Conf. on Communication, Control and Computing, Urbana, Illinois, 1983, pp. 224-233.
- [5] M. R. FELLOWS AND M. A. LANGSTON, *Nonconstructive advances in polynomial-time complexity*, Inform. Process. Lett., 26 (1987), pp. 157-162.
- [6] ———, *Fast self-reduction algorithms for combinatorial problems of VLSI design*, in Proc. 3rd Aegean Workshop on Computing, Corfu, Greece, 1988, pp. 278-287.
- [7] ———, *Layout permutation problems and well-partially-ordered sets*, in Proc. 5th MIT Conf. on Advanced Research in VLSI, Cambridge, Massachusetts, 1988, pp. 315-327.
- [8] ———, *Nonconstructive tools for proving polynomial-time decidability*, J. Assoc. Comput. Mach., 35 (1988), pp. 727-739.
- [9] ———, *On search, decision and the efficiency of polynomial-time algorithms*, in Proc. 21st ACM Symp. on Theory of Computing, Seattle, Washington, 1989, pp. 501-512.
- [10] M. R. GAREY, R. L. GRAHAM, D. S. JOHNSON, AND D. E. KNUTH, *Complexity results for bandwidth minimization*, SIAM J. Appl. Math., 34 (1978), pp. 477-495.
- [11] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, CA, 1979.
- [12] E. M. GURARI AND I. H. SUDBOROUGH, *Improved dynamic programming algorithms for bandwidth minimization and the min cut linear arrangement problem*, J. Algorithms, 5 (1984), pp. 531-546.
- [13] T. LENGAUER, *Black-white pebbles and graph separation*, Acta Inform., 16 (1981), pp. 465-475.
- [14] W. MADER, *Hinreichende Bedingungen für die Existenz von Teilgraphen, die zu einem vollständigen Graphen homöomorph sind*, Math. Nachr., 53 (1972), pp. 145-150.
- [15] ———, *Wohlquasi geordnete Klassen endlicher Graphen*, J. Combin. Theory Ser. B, 12 (1972), pp. 105-122.

- [16] F. S. MAKEDON AND I. H. SUDBOROUGH, *On minimizing width in linear layouts*, Lecture Notes in Comput. Sci., 154 (1983), pp. 478–490.
- [17] Z. MILLER AND I. H. SUDBOROUGH, *Polynomial algorithms for recognizing small cutwidth in hypergraphs*, in Proc. 2nd Aegean Workshop on Computing, Loutraki, Greece, 1986, pp. 252–260.
- [18] C. H. PAPADIMITRIOU, private communication, 1988.
- [19] T. D. PARSONS, *Pursuit-evasion in a graph*, in Theory and Application of Graphs, Y. Alavi and D. R. Lick, eds., Springer-Verlag, Berlin, New York, 1976, pp. 426–441.
- [20] N. ROBERTSON, private communication, 1987.
- [21] N. ROBERTSON AND P. D. SEYMOUR, *Graph minors IV. Tree-width and well-quasi-ordering*, J. Combin. Theory Ser. B, 48 (1990), pp. 227–254.
- [22] ———, *Graph minors V. Excluding a planar graph*, J. Combin. Theory Ser. B, 41 (1986), pp. 92–114.
- [23] ———, *Graph minors X. Obstructions to tree-decomposition*, J. Combin. Theory Ser. B, 52 (1991), pp. 153–190.
- [24] ———, *Graph minors XIII. The disjoint paths problem*, to appear.
- [25] ———, *Graph minors XVI. Wagner's conjecture*, to appear.
- [26] P. D. SEYMOUR, private communication, 1989.
- [27] I. H. SUDBOROUGH, private communication, 1988.

# Correspondence

## Small Diameter Symmetric Networks from Linear Groups

Lowell Campbell, Gunnar E. Carlsson, Michael J. Dinneen, Vance Faber, Michael R. Fellows, Michael A. Langston, James W. Moore, Andrew P. Mullhaupt, and Harlan B. Sexton

**Abstract**—In this note is reported a collection of constructions of symmetric networks that provide the largest known values for the number of nodes that can be placed in a network of a given degree and diameter. Some of the constructions are in the range of current potential engineering significance. The constructions are Cayley graphs of linear groups obtained by experimental computation.

**Index Terms**—Cayley graphs, interconnection networks, linear groups.

### I. INTRODUCTION

The problem of constructing large graphs of a given degree and diameter has received much attention, and is significant for parallel processing because it models two important constraints in the design of massively parallel processing systems: 1) there are limits on the number of processors to which any processor in the network can be directly connected, and 2) the distance between any two processors in the network should not be too great. Other applications of such networks include shared-key cryptographic protocols and the design of local area networks. See [3] and [9] for recent surveys.

In this paper we give evidence that the table of largest known constructions for small values of the two parameters can be improved for many parameter values by methods based on finite linear groups. In many cases the networks we describe here are dramatically larger than those previously known.

Many of our improvements are in the range of the numbers of processors currently being considered for large parallel processing systems, suggesting that some of these constructions may merit further investigation for such applications. This is the focus of continuing research by some of our party. In this note we present only our accumulated results on the now classic problem of network construction. In particular, we do not address the many interesting problems concerning routing and data exchange that would be crucial for most parallel processing applications.

Manuscript received January 12, 1989; revised December 15, 1990. This work was supported in part by the National Science Foundation under Grant MIP-8693879, by the Office of Naval Research under Contracts N00014-88-K-0343 and N00014-88-K-0546, and by the National Aeronautics and Space Administration under Grant NAGW-1406. A preliminary version of this paper was presented at the Second Symposium on the Frontiers of Massively Parallel Computation, Fairfax, VA. Inquiries should be directed to M. Fellows.

L. Campbell is with the Department of Electrical Engineering, University of Idaho, Moscow, ID 83843.

G. E. Carlsson is with the Department of Mathematics, Princeton University, Princeton, NJ 08540.

M. J. Dinneen and M. R. Fellows are with the Department of Computer Science, University of Victoria, Victoria, BC, Canada V8W 3P6.

V. Faber and J. W. Moore are with Los Alamos National Laboratories, Los Alamos, NM 87545.

M. A. Langston is with the Department of Computer Science, University of Tennessee, Knoxville, TN 37996.

A. P. Mullhaupt is with the Department of Mathematics, University of New Mexico, Albuquerque, NM 87131.

H. B. Sexton is with Lucid, Inc., Menlo Park, CA 94025.

IEEE Log Number 9101691.

For an overview of our results see Table I, which represents an updated version, obtained from Bermond [5] of the table published in [3]. Interested readers are advised that a "current" table incorporating the results of many workers on this problem is maintained by and available from that helpful source. The entries in the table that are due to our efforts and reported on in this note are marked in bold. Other entries that have been obtained by Cayley graph techniques are marked with an asterisk. In particular, two other groups of researchers have recently and independently obtained record-breaking constructions based on linear groups [4], [8].

### II. ALGEBRAIC SYMMETRY AS AN ORGANIZING PRINCIPLE FOR PARALLEL PROCESSING

There are important considerations apart from degree and diameter that must figure in any choice of network topology for parallel computation. A network is (*vertex*-) *symmetric* if for any two nodes  $u, v$  there is an automorphism of the network mapping  $u$  to  $v$ . Our approach yields symmetric constructions, and we believe that in this may lie their greater value. Symmetry is one of the most powerful and natural tools to apply to the central problem of massively parallel computation: how to organize and coordinate computational resources.

The symmetries of the networks we describe are represented by simple algebraic operations (such as  $2 \times 2$  matrix multiplications and modulo arithmetic). The main advantage of algebraically constructed networks is that the developed mathematical resources of algebra are available to structure the problems of

- 1) design and description
- 2) testing
- 3) data exchange and routing
- 4) scheduling and computation mapping.

The appeal of hypercubes, cube-connected cycles, butterfly networks, and others rests in large part on the availability of easily computed (and comprehended) symmetries. These popular network designs and those that we describe all belong to a class of algebraic networks based on vector spaces and their symmetry groups. For recent algebraic approaches to routing algorithms, deadlock avoidance, emulation, and scheduling for algebraically described networks of this kind see [2], [1], [8], [11], and [12].

Our main result in this brief paper is a demonstration that algebraic symmetry provides a powerful approach to problem 1), design and description. Our approach centers on the following definition.

**Definition:** If  $A$  is a group and  $S \subseteq A$  is a generating set that is closed under inverses, i.e.,  $S = S \cup S^{-1}$ , then the (undirected) *Cayley graph*  $(A, S)$  is the graph with vertex set  $A$  and with an edge between elements  $a$  and  $b$  of  $A$  if and only if  $as = b$  for some  $s \in S$ .

Every Cayley network is symmetric (symmetries are given by group multiplication). The degree of a Cayley graph  $(A, S)$  is  $\Delta = |S|$  and the diameter of  $(A, S)$  is

$$D = \max_{a \in A} \left\{ \min_t : a = s_1 \cdots s_t, s_i \in S, i = 1, \dots, t \right\}.$$

It is remarkable (but, indeed, natural) that most networks that have been considered for large parallel processing systems (including hypercubes, torus grids, cube-connected-cycles and butterfly networks)

TABLE I  
ALGEBRAIC SYMMETRY AS AN ORGANIZING PRINCIPLE FOR PARALLEL PROCESSING

$\Delta$	$D$								
	2	3	4	5	6	7	8	9	10
3	10	20	38	70	128	184	320	540	938
4	15	40	95	364	734	1081*	2943*	7439*	15657*
5	24	70	182	532	2742	4368	11200	33600	123120
6	32	105	355	1081	7832	13310	50616	202464	682080
7	50	128	506	2162	10554	39732	140000	911088	2002000
8	57	203	842	3081	39258	89373	455544	1822175	3984120
9	74	585	1248	6072	74954	215688	910000	3019632	15686400
10	91	650	1820	12144	132932	486837	2002000	7714494	47059200

are Cayley graphs. A standard reference on Cayley graphs is [7]. For a Cayley graph description of the cube-connected-cycles see [10].

Symmetry immediately provides the following advantage for the design problem considered here: to compute the diameter of a Cayley graph it is only necessary to compute the distances from a single node to all others. Furthermore, the compactness of an algebraic description allows for an efficient computational search strategy.

Our results were obtained by experimental computing with relatively simple programs on small machines (an IBM PC and a VAX 11/780). The programs followed closely the above expression for the diameter of a Cayley graph. Having focused (by setting the appropriate program parameters) on a particular kind of matrix group, and on a choice of cardinality for the generating set (hence the degree of the resulting graph), the diameter was computed for repeated random choices of the generating set until (in the favorable case) a new record was obtained. Consonant with the above expression for the diameter, this is done by starting with the identity of the group as the *live* set, multiplying the elements of the live set with the elements of the generator set, recording any new elements obtained (the new live set) in a large array representing all elements in the target group, and repeating this until no new elements are obtained. The number of repetitions until this occurs is the diameter.

The reader may reasonably wonder about several things, beginning with the large number of authors of this note and including perhaps the question of whether some voodoo was employed in choosing the target groups and in exploring the search space of generator sets. The explanation of the first is simply that exploration of this approach to this design problem has continued among us at a low level for a number of years beginning with the seminal work of the author subset: Carlsson and Sexton. Although we have tried several "sophisticated" heuristics for choosing groups and generators, we must honestly report that none of these has proven better than simple and straightforward random search, with the exception of the nearly obvious guidance that one should choose a nonabelian group! Several of our record-breaking constructions employ upper-triangular matrix groups, but we are unable to explain why these worked better than other possibilities.

Thus, in some sense these results are less interesting than one might at first suppose, although the above information may underscore our main point: the power of an algebraic approach (even a simple one). A sophisticated understanding of what is possible by the method of Cayley graphs would be highly desirable, but it seems to present a difficult mathematical problem.

The next section describes some examples of our constructions and the associated costs of our computational explorations.

### III. EXAMPLE CONSTRUCTIONS

Given that a "winning" set of generators exists for a group it would be interesting to know the expected time for random search to

discover a winning set. We have no real information on this (it would seem to be a difficult mathematical problem to give any bounds), but we do indicate in the example descriptions that follow the time that was required for the particular search that uncovered the construction as a rough indication of the amount of computational effort involved. About half of the record-breaking constructions that we report here (the ones of smaller order!) were obtained on a PC, by a search program running in some cases for only a few minutes and in some cases for a few days. For the approach that we have taken memory is a more important computational bottleneck than speed.

In what follows  $GL[n, q]$  denotes the (*general linear*) group of  $n \times n$  matrices with entries in the finite field with  $q$  elements (since below  $q$  is always a prime, this is just the integers mod  $q$ ), and  $SL[n, q]$  is the *special linear* subgroup of  $GL[n, q]$  consisting of those matrices with determinant 1.

*Example 1:* Degree 5, diameter 7: 4368 vertices.

This is a Cayley graph on the subgroup of  $GL[2, 13]$  consisting of the matrices with determinant in the set  $\{1, -1\}$ . The generators are the following elements together with their inverses.

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \text{ order 2} \quad \begin{bmatrix} 11 & 2 \\ 8 & 12 \end{bmatrix} \text{ order 52} \quad \begin{bmatrix} 11 & 4 \\ 7 & 5 \end{bmatrix} \text{ order 14.}$$

The discovery time for this construction was approximately 10 hours on an IBM PC for a small Pascal program.

*Example 2:* Degree 8, diameter 7: 89373 vertices

This is a Cayley graph on a subgroup of  $GL[3, 31]$ . The generators are the following elements together with their inverses.

$$\begin{bmatrix} 1 & 12 & 10 \\ 0 & 1 & 15 \\ 0 & 0 & 25 \end{bmatrix} \text{ order 93} \quad \begin{bmatrix} 1 & 25 & 15 \\ 0 & 1 & 4 \\ 0 & 0 & 5 \end{bmatrix} \text{ order 93}$$

$$\begin{bmatrix} 1 & 29 & 29 \\ 0 & 1 & 16 \\ 0 & 0 & 5 \end{bmatrix} \text{ order 93} \quad \begin{bmatrix} 1 & 27 & 5 \\ 0 & 1 & 8 \\ 0 & 0 & 1 \end{bmatrix} \text{ order 31}$$

The discovery time for this construction was approximately 3 hours of CPU time on a VAX 11/780.

*Example 3:* Degree 10, diameter 5: 12144 vertices.

This is a Cayley graph on the group  $SL[2, 23]$ . The generators are the following elements together with their inverses.

$$\begin{bmatrix} 9 & 0 \\ 18 & 18 \end{bmatrix} \text{ order 11} \quad \begin{bmatrix} 13 & 10 \\ 18 & 21 \end{bmatrix} \text{ order 11} \quad \begin{bmatrix} 9 & 10 \\ 0 & 17 \end{bmatrix} \text{ order 22}$$

$$\begin{bmatrix} 14 & 7 \\ 19 & 3 \end{bmatrix} \text{ order 22} \quad \begin{bmatrix} 18 & 13 \\ 17 & 20 \end{bmatrix} \text{ order 24.}$$

The discovery time for this construction was approximately 2 hours on an IBM PC.

TABLE II

Parameters	Order	Group	Generators: order $S = S U S^{-1}$
degree 5 diameter 7	4368	subgroup of GL[2,13]	[0,1,1,0]:2 [11,4,7,5]:14 [11,2,8,12]:52
degree 5 diameter 8	8788	subgroup of GL[3,13]	[1,0,4,0,1,0,0,0,12]:2 [1,5,6,0,1,9,0,0,5]:52 [1,2,12,0,1,8,0,0,5]:52
degree 5 diameter 9	25308	PSL[2,37]	[0,36,1,0]:2 [34,26,34,1]:37 [2,16,11,33]:37
degree 5 diameter 10	123120	GL[2,19]	[0,1,1,0]:2 [11,16,0,15]:18 [16,11,2,0]:45
degree 6 diameter 4	355	subgroup of GL[2,71]	[54,66,0,1]:5 [5,43,0,1]:5 [57,38,0,1]:5
degree 6 diameter 5	1081	subgroup of GL[2,47]	[7,20,0,1]:23 [6,33,0,1]:23 [9,42,0,1]:23
degree 6 diameter 7	13310	subgroup of GL[3,11]	[1,2,7,0,1,0,0,0,10]:22 [1,5,2,0,1,2,0,0,4]:55 [1,6,10,0,1,3,0,0,5]:55
degree 6 diameter 8	50616	SL[2,37]	[32,24,35,2]:19 [23,16,28,34]:36 [12,24,15,27]:37
degree 6 diameter 9	202464	subgroup of GL[2,37]	[25,1,31,1]:36 [12,35,23,30]:76 [12,4,28,16]:152
degree 6 diameter 10	682080	GL[2,29]	[28,10,8,8]:28 [17,13,16,27]:28 [3,4,27,14]:840
degree 7 diameter 4	506	subgroup of GL[2,23]	[22,1,0,1]:2 [13,16,0,1]:11 [3,16,0,1]:11 [19,12,0,1]:22
degree 7 diameter 5	2162	subgroup of GL[2,47]	[46,1,0,1]:2 [4,20,0,1]:23 [20,27,0,1]:46 [29,14,0,1]:46
degree 7 diameter 7	39732	PSL[2,43]	[0,42,1,0]:2 [18,16,38,41]:22 [34,2,37,6]:22 [8,28,14,33]:43
degree 7 diameter 8	101232	subgroup of GL[2,37]	[0,1,1,0]:2 [21,34,17,17]:6 [21,1,4,2]:9 [27,26,4,8]:74
degree 7 diameter 9	911088	subgroup of GL[2,37]	[0,1,1,0]:2 [23,17,14,26]:18 [25,16,13,6]:36 [27,33,19,22]:684
degree 7 diameter 10	1822176	GL[2,37]	[0,1,1,0]:2 [1,19,14,16]:17 [36,1,12,0]:18 [35,28,34,12]:456
degree 8 diameter 3	203	subgroup of GL[2,29]	[16,9,0,1]:7 [16,21,0,1]:7 [25,15,0,1]:7 [25,9,0,1]:7
degree 8 diameter 4	812	subgroup of GL[2,29]	[12,1,0,1]:4 [20,24,0,1]:7 [6,27,0,1]:14 [15,18,0,1]:28
degree 8 diameter 5	3081	subgroup of GL[2,79]	[46,43,0,1]:13 [49,72,0,1]:39 [19,26,0,1]:39 [13,13,0,1]:39
degree 8 diameter 7	89373	subgroup of GL[2,31]	[1,4,25,0,1,23,0,0,1]:31 [1,29,29,0,1,16,0,0,5]:93 [1,12,10,0,1,15,0,0,25]:93
			[1,6,17,0,1,24,0,0,5]:93
degree 8 diameter 8	455544	subgroup of GL[2,37]	[21,9,17,5]:57 [0,26,3,1]:171 [28,32,33,33]:171 [9,34,25,16]:342
degree 8 diameter 9	1822176	GL[2,37]	[12,13,34,33]:18 [36,6,20,10]:36 [35,3,19,35]:684 [26,10,36,31]:1368
degree 9 diameter 5	6072	PSL[2,23]	[0,22,1,0]:2 [2,18,4,2]:11 [10,1,21,16]:24 [6,19,4,9]:24 [22,0,1,22]:46
degree 9 diameter 8	682080	GL[2,29]	[0,1,1,0]:2 [5,22,18,26]:14 [17,15,21,4]:840 [2,5,10,21]:840 [23,12,11,21]:840
degree 10 diameter 5	12144	SL[2,23]	[9,0,18,18]:11 [13,10,18,21]:11 [9,10,0,17]:22 [14,7,19,3]:22 [18,13,17,20]:24
degree 10 diameter 8	1822176	subgroup of GL[2,37]	[21,12,22,5]:57 [9,12,6,26]:456 [35,10,17,32]:684 [5,31,35,14]:684 [11,3,33,7]:1368

## IV. THE CONSTRUCTIONS

During the publication process for this note we have become aware of a new approach to this design problem, not based on Cayley graphs, that shares with our approach the aspects of 1) a significant exploitation of symmetry, and 2) computational exploration [5]. This has had the effect on this note of removing from "bold" seven entries of the original version of Table I. We have retained the descriptions of the Cayley graphs that gave those entries in the table that follows, as they may still be of interest by virtue of their vertex symmetry or other properties.

## V. CONCLUSIONS

Our main contribution in this brief presentation is the demonstration of the power of an algebraic approach to the problem of constructing large networks of a given degree and diameter. The success of the relatively limited search we have so far conducted seems to indicate that further exploration based on Cayley graphs may be productive. Major problems relevant to applicants in parallel processing and not addressed here concern message routing and data exchange. Solutions are likely to be much more complicated in such networks as we have described than in the familiar (Cayley graph) networks of hypercubes and cube-connected cycles, and this remains an area for further research.

## REFERENCES

- [1] F. Annexstein, M. Baumslag, and A.L. Rosenberg, "Group action graphs and parallel architectures," *COINS Tech. Rep. 87-133*, Univ. of Massachusetts, Amherst, MA, 1987.
- [2] S.B. Akers and B. Krishnamurthy, "On group graphs and their fault-tolerance," *IEEE Trans. Comput.*, vol. C-36, pp. 885-888, 1987.
- [3] J.C. Bermond, C. Delorme, and J.J. Quisquater, "Strategies for interconnection networks: Some methods from graph theory," *J. Parallel and Distributed Comput.*, vol. 3, pp. 433-449, 1986.
- [4] J. Bond, C. Delorme, and W.F. de La Vega, "Large Cayley graphs with small degree and diameter," *Rapport de Recherche no. 392*, LRI, Orsay, France, 1987.
- [5] J.C. Bermond, private communication, June 1989.
- [6] R. Bar-Yehuda and T. Etzion, "Connections between two cycles—A new design of dense processor interconnection networks," *Tech. Rep. 528*, Dep. Comput. Sci., Technion, Haifa, Israel, 1988.
- [7] N. Biggs, *Algebraic Graph Theory*. London, England: Cambridge University Press, 1974.
- [8] D.V. Chudnovsky, G.V. Chudnovsky, and M.M. Denneau, "Regular graphs with small diameter as models for interconnection networks," in *Proc. Third Int. Conf. Supercomput.*, Boston, MA, May 1988, pp. 232-239.
- [9] F.R.K. Chung, "Diameters of graphs: Old problems and new results," *Congressus Numerantium* 60, pp. 295-317, 1987.
- [10] G.E. Carlsson, J.E. Cruthirds, H.B. Sexton, and C.G. Wright, "Interconnection networks based on a generalization of cube-connected cycles," *IEEE Trans. Comput.*, vol. C-34, pp. 769-777, 1985.
- [11] V. Faber, "Global communication algorithms for hypercubes and other Cayley coset graphs," *Tech. Rep.*, Los Alamos National Lab., Los Alamos, NM, 1988.
- [12] M. Fellows, "A category of graphs for parallel processing," *Tech. Rep. Dep. Comput. Sci., Univ. of Idaho, Moscow, ID*, 1988.

# Constructive complexity\*

Karl Abrahamson

*Department of Computer Science, Washington State University, Pullman, WA 99164, USA*

Michael R. Fellows

*Department of Computer Science, University of Victoria, Victoria, B.C., Canada V8W 3P6*

Michael A. Langston

*Department of Computer Science, University of Tennessee, Knoxville, TN 37996, USA*

Bernard M.E. Moret

*Department of Computer Science, University of Victoria, Victoria, B.C. Canada V8W 3P6*

Received 10 June 1989

## Abstract

Abrahamson, K., M.R. Fellows, M.A. Langston and B.M.E. Moret, Constructive complexity, Discrete Applied Mathematics 34 (1991) 3-16.

Powerful and widely applicable, yet inherently nonconstructive, tools have recently become available for classifying decision problems as solvable in polynomial time, as a result of the work of Robertson and Seymour. These developments challenge the established view that equates tractability with polynomial-time solvability, since the existence of an inaccessible algorithm is of very little help in solving a problem. In this paper, we attempt to provide the foundations for a constructive theory of complexity, in which membership of a problem in some complexity class indeed implies that we can find out how to solve that problem within the stated bounds. Our approach is based on relations, rather than on sets; we make much use of self-reducibility and oracle machines, both conventional and "blind", to derive a series of results which establish a structure similar to that of classical complexity theory, but in which we are in fact able to prove results which remain conjectural within the classical theory.

## 1. Introduction

Powerful and widely applicable, yet inherently nonconstructive, tools have recently become available for classifying decision problems as solvable in poly-

\* This research is supported in part by the Washington State Technology Center, by the National Science Foundation under grant MIP-8703879, by the Office of Naval Research under contracts N00014-88-K-0343 and N00014-88-K-0456, and by the NSERC of Canada.

nomial time, as a result of the work of Robertson and Seymour [24, 25] (see also [12]). When applicable, the combinatorial finite basis theorems at the core of these developments are nonconstructive on two distinct levels. First, a polynomial-time algorithm is only shown to *exist*: no effective procedure for finding the algorithm is established. Secondly, even if known, the algorithm only *decides*: it uncovers nothing like “natural evidence”. Examples of the latter had been rare; one of the few such is primality testing, where the existence of a factor can be established without, however, yielding a method for finding said factor. Moreover, there had been a tendency to assume that there is no significant distinction between decision and construction for sequential polynomial time [13]. However, there is now a flood of polynomial-time algorithms based on well-partial-order theory that produce no natural evidence [4, 5, 20]. For example, determining whether a graph has a knotless embedding in 3-space (i.e., one in which no cycle traces out a nontrivial knot) is decidable in cubic time [20], although no *recursive* method is known for producing such an embedding even when one is known to exist. What is worse (from a computer scientist’s point of view), the nonconstructive character of these theorems is inherent, as they are independent of standard theories of arithmetic [8]; thus attempts at “constructivizing” these results [7] may yield practical algorithms in some cases, but cannot succeed over the entire range of application.

These developments cast a shadow on the traditional view that equates the tractability of a problem with the existence of a polynomial-time algorithm to solve the problem. This view was satisfactory as long as existence of such algorithms was demonstrated constructively; however, the use of the existential quantifier is finally “catching up” with the algorithm community. The second level of nonconstructiveness—the lack of natural evidence—has been troubling theoreticians for some time: how does one trust an algorithm that only provides one-bit answers? Even when the answer is uniformly “yes”, natural evidence may be hard to come by. (For instance, we know that all planar graphs are 4-colorable and can check planarity in linear time, but are as yet unable to find a 4-coloring in linear time.) Robertson and Seymour’s results further emphasize the distinction between deciding a problem and obtaining natural evidence for it.

All of this encourages us to consider ways in which some of the foundations of complexity theory might be alternatively formulated from a *constructive* point of view. (We intend the term “constructive” to convey an informal idea of our goals; this is not to be confused with a *constructivist* approach [2], which would certainly answer our requirements, but may well prove too constrictive. However, our intent is certainly constructivist, in that we intend to substitute for simple existence certain acceptable styles of proof based on “natural” evidence.)

We base our development on the relationships between the three aspects of a decision problem: evidence checking, decision, and searching (or evidence construction). The heart of our constructive formulation is to provide for each problem to be equipped with its own set of *allowable* proofs, in much the same manner as the usual definition of NP-membership [9]; moreover, in deterministic classes, the

proof itself should be constructible. This leads us to a formalization based on evidence generators and evidence checkers—exactly as in classical complexity theory, but where the checker is given as part of the problem specification rather than discovered as part of the solution. In other words, we come to the “Algorithm Shop” prepared to accept only certain types of evidence.

The result of this approach is a formulation based on *relations*, rather than on sets as in the classical formulation. This formulation provides a natural perspective on self-reducibility and oracle complexity [1, 14, 27–29], concepts which have recently received renewed scrutiny. We define oracle mechanisms through which we can ask interesting questions about the value of proofs and the nature of nonconstructiveness; we manage to answer some of these questions, including one which remains a conjecture in its classical setting.

## 2. Problems and complexity classes

In classical complexity theory, a decision problem is a language (or set) over some alphabet,  $L \subseteq \Sigma^*$ ; the main question concerning a language is the *decision* problem: given some string  $x$ , is it an element of the language  $L$ ? However, since a simple “yes” or “no” answer clearly lacks credibility, one may require that the algorithm also produce a proof for its answer, within some prespecified acceptability criteria. Such a proof—or sketch of one—is termed *evidence* and the problem of deciding membership as well as producing evidence is called a *search* (or certificate construction) problem. Finally, in order for such a proof to be of use, it must be concise and easily checked; the problem of verifying a proof for some given string is the *checking* problem. In a relational setting, all three versions admit a particularly simple formulation; for a fixed relation  $\mathcal{R} \subseteq \Sigma^* \times \Sigma^*$ :

- *Checking*: given  $(x, y)$ , does  $(x, y) \in \mathcal{R}$ ?
- *Deciding*: given  $x$ , does there exist a  $y$  such that  $(x, y) \in \mathcal{R}$ ?
- *Searching*: given  $x$ , find a  $y$  such that  $(x, y) \in \mathcal{R}$ .

For the most part, the decision and search versions of a problem have been held to be of comparable complexity, as such is indeed the case for many problems (witness the notion of NP-completeness and NP-equivalence); as to checking, classical complexity theory has only considered it in the case of nondeterministic classes. Schnorr [27] has studied the relationship between the decision and search versions of a problem and attempted to characterize this relationship for decision problems within NP; Schöning [28] has proposed a model of computation (*robust* oracle machines) under which decision automatically incorporates checking; other authors have also investigated the search version of decision problems and proposed mechanisms for its characterization [1, 14, 17, etc.]. We go one step further and (essentially) ignore the decision version, concentrating instead on the checking and search versions and their relationship.

**Definition 2.1.** A decision problem is a pair  $\Pi = (I, M)$ , where  $M$  is a checker and  $I$  is the set of “yes” instances.

The checker  $M$  defines a relation between yes instances and acceptable evidence for them, providing a concise and explicit description of the relation. Moreover, this formulation only allows problems for which a checker can be specified, thereby avoiding existence problems. However, in the following, we shall generally use the relational formalism explicitly; in that formalism, given relation  $\mathcal{R}$ , the set of acceptable instances of the problem is the domain of the relation, which we denote  $D(\mathcal{R})$ .

This definition makes a problem into a subjective affair: two different computer scientists may come to the “Algorithm Shop” with different checkers and thus require different evidence-generating algorithms—to the point where one may be satisfied with a constant-time generator and the other may require an exponential-time one. Many classical problems (such as the known NP-complete problems) have “obvious” evidence—what we shall call *natural evidence*; for instance, the natural evidence for the satisfiability problem is a satisfying truth assignment.

Solving such a problem entails two steps: generating suitable evidence and then checking the answer with the help of the evidence; this sequence of steps is our constructive version of the classical decision problem. (Indeed, to reduce our version to the classical one, just reduce the checker to a trivial one which simply echoes the first bit of the evidence string.) Thus the complexity of such problems is simply the complexity of their search and checking components, which motivates our definition of complexity classes.

**Definition 2.2.** A constructive complexity class is a pair of classical complexity classes,  $(C_1, C_2)$ , where  $C_1$  denotes the resource bounds within which the evidence generator must run and  $C_2$  the bounds for the checker. Resource bounds are defined with respect to the classical statement of the problem, i.e., with respect to the size of the domain elements; for nondeterministic classes,  $C_1$  is omitted, thereby denoting that the evidence generator may simply guess the evidence.

For instance, we shall define the class  $P_c$  to be the pair  $(P, P)$ , thus requiring both generator and checker to run in polynomial time; in other words,  $P_c$  is the class of all P-checkable and P-searchable relations. In contrast, we shall define  $NP_c$  simply as the class of all P-checkable relations, placing no constraints on the generation of evidence. (But note that, since the complexity of checking is defined with respect to the domain of the relation, the polynomial-time bound on checking translates into a polynomial-time bound on the length of acceptable evidence.)

**Definition 2.3.** A problem  $(I, M)$  belongs to a class  $(C_1, C_2)$  if and only if the relation defined by  $M$  on  $I$  is both  $C_1$ -searchable and  $C_2$ -checkable.

These general definitions only serve as guidelines in defining interesting constructive complexity classes. Counterparts of some classical complexity classes immediately suggest themselves: since all nondeterministic classes are based on the existence of checkable evidence, they fit very naturally within our framework. Thus, for instance, we define

- $\text{NLOGSPACE}_c = (-, \text{LOGSPACE})$ ,
- $\text{NP}_c = (-, \text{P})$ ,
- $\text{NEXP}_c = (-, \text{EXP})$ .

(Note that, even if  $\text{NEXP} = \text{EXP}$ —i.e.,  $\text{NEXP}$  is  $\text{EXP}$ -decidable—, it does not follow that  $\text{NEXP}$  is  $\text{EXP}$ -searchable; indeed, there exists a relativization where the first statement is true but the second false [11]. In contrast,  $\text{NP}$  is  $\text{P}$ -decidable if and only if it is  $\text{P}$ -searchable. This contrast—an aspect of upward separation—adds interest to our definitions of  $\text{NP}_c$  and  $\text{NEXP}_c$ .) Deterministic classes, on the other hand, may be characterized by giving generator and checker the same resource bounds:

- $\text{LOGSPACE}_c = (\text{LOGSPACE}, \text{LOGSPACE})$ ,
- $\text{P}_c = (\text{P}, \text{P})$ ,
- $\text{PSPACE}_c = (\text{PSPACE}, \text{PSPACE})$ .

The principal tool in the study of complexity classes is the reduction. Many-one reductions between decision problems are particularly simple in the classical framework: one simply maps instances of one problem into instances of the other, respecting the partition into yes and no instances, so that the original instance is accepted if and only if the transformed one is. However, in our context, such reductions are both insufficient and too stringent: we require evidence to go along with the “answer”, but can also use this evidence to “correct any error” made during the forward transformation. In essence, the goal of a constructive reduction is to recover evidence for the problem at hand from the evidence gleaned for the transformed instance. Thus a many-one reduction between two problems is given by a *pair* of maps; similar mechanisms have been proposed by Levin [17] for transformations among search problems and by Krentel [15] (who calls his “metric reductions”) for transformations among optimization problems, where the solution is the value of the optimal solution. Note that the strict preservation of the partition into yes and no instances is no longer required: we must continue to map yes instances into yes instances, but can also afford to map no instances into yes instances, as we shall be able to invalidate the apparent “yes” answer when attempting to check the evidence.

**Definition 2.4.** A constructive (many-one) transformation from problem (relation)  $\mathcal{R}_1$  to problem  $\mathcal{R}_2$  is a pair of functions  $(f, g)$ , such that

- (1)  $x \in D(\mathcal{R}_1) \Rightarrow f(x) \in D(\mathcal{R}_2)$ ; and
- (2)  $x \in D(\mathcal{R}_1) \wedge (f(x), y') \in \mathcal{R}_2 \Rightarrow (x, g(x, y')) \in \mathcal{R}_1$ .

(Note that, for obvious reasons, the evidence-transforming map,  $g$ , must take the

original instance as argument as well as the evidence for the transformed instance.) With each complexity class we associate a suitable class of transformations by bounding the resources available for the computation of the two maps: for instance, transformations within  $NP_c$  must run in polynomial time and transformations within  $P_c$  or  $NLOGSPACE_c$  in logarithmic space.

**Theorem 2.5.** *Constructive many-one reductions are transitive.*

For both resource bounds mentioned above, the proof is trivial.

Equipped with many-one reductions, we can proceed to define, in the obvious way, a notion of completeness for our classes. Note that a desirable consequence of our definitions would be that a problem complete for some class in the classical setting remains complete for the constructive version of the class when equipped with the natural checker, if available. Since the notion of natural checker is rather vague, we establish this result for some specific classes.

Let SAT/Nat be the satisfiability problem (in conjunctive normal form) equipped with the natural checker which requires a truth assignment as evidence; similarly, let CV/Nat be the circuit value problem (see [16]) equipped with evidence consisting of the output of each gate and let GR/Nat be the graph reachability problem (see [26]) equipped with evidence consisting of the sequence of edges connecting the two endpoints.

**Theorem 2.6.** (1) SAT/Nat is  $NP_c$ -complete.

(2) CV/Nat is  $P_c$ -complete.

(3) GR/Nat is  $NLOGSPACE_c$ -complete.

**Proof.** We only sketch the proof of the first assertion; the others use a similar technique, taking advantage of the fact that the generic reductions used in the classical proofs are constructive.

That SAT/Nat is in  $NP_c$  is obvious. Denote by  $\phi(M, x)$  the formula produced by Cook's transformation when run on a polynomial-time nondeterministic Turing machine  $M$  and input string  $x$ . Now let  $\Pi$  be some arbitrary problem in  $NP_c$ ; then there exists some evidence generator for  $\Pi$ , which can be given by a polynomial-time nondeterministic Turing machine  $M'$ . Let  $f(x) = \phi(M', x)$  and let  $g(x, y')$  be the output produced by  $M'$  in the computation described by the truth assignment  $y'$  for the formula  $\phi(M', x)$ . Then the pair  $(f, g)$  is easily seen to be a constructive, many-one, polynomial-time reduction from  $\Pi$  to SAT/Nat.  $\square$

The polynomial-time reductions found in the NP-completeness literature are generally constructive, in terms of natural evidence. Thus, for example, the vertex cover problem with natural evidence (namely, the vertex cover) is  $NP_c$ -complete. Similar comments apply for P-complete problems and  $NLOGSPACE_c$ -complete problems. A natural question at this point is whether or not the following statement

holds, for a classical complexity class  $C$  and its constructive counterpart  $C_c$ :

$\Pi$  is  $C_c$ -complete if and only if  $\Pi \in C_c$  and  $D(\Pi)$  is  $C$ -complete.

The only if part would obviously hold if we had required our constructive reductions to preserve the partition between yes and no instances. The if part can be interpreted as follows in the case of  $C = NP$ . We know that weak probabilistic evidence exists for all problems in  $NP$ , as a form of zero-knowledge proof exists for all such sets [3,10]; we also suspect that, for some sets in  $NP$  (such as the set of composite numbers), some forms of evidence (factors in our example) are harder to find than others (e.g., witnesses of the type used by Rabin [22]). Thus the if part would imply that there exists weak deterministic evidence for membership in an  $NP$ -complete set.

### 3. Self-reducibility and oracle complexity

Constructive complexity is closely tied to the issue of self-reducibility. Self-reducibility itself plays an important role in classical complexity theory (see, e.g. [1]), but lacks a natural definition in that framework (whence the large number of distinct definitions). In our constructive formulation, however, self-reducibility has a very natural definition, which, moreover, very neatly ties together the three facets of a decision problem.

**Definition 3.1.** A problem  $\Pi$  in some class  $(C_1, C_2)$  is (Turing) self-reducible if it is  $C_2$ -searchable with the help of an oracle for  $D(\Pi)$ .

In other words, a problem self-reduces if it can be searched as fast as it can be checked with the help of a decision oracle; all three facets of a decision problem indeed come together in this definition. In the case of  $NP_c$  problems, our definition simply states that such problems self-reduce if they can be solved in polynomial time with the help of an oracle for their decision version; such a definition coincides with the self-1-helpers of Ko [14] and the self-computable witnesses of Balcazar [1].

A natural question to ask about reducibility is whether  $D(\Pi)$  is really an appropriate oracle for  $\Pi$ : would not a more powerful oracle set make the search easier? We can show that such is not the case and that, in fact, our choice of oracle is in some sense optimal.

**Definition 3.2.**  $\Pi$  has *oracle complexity* at most  $f(n)$  if there is a deterministic oracle algorithm, using any fixed oracle, that makes at most  $f(n)$  oracle queries on inputs of length  $n$  and produces acceptable evidence for  $\Pi$ .

We restrict the oracle algorithm to run within appropriate resource bounds: such as polynomial time for problems in  $NP_c$  and logarithmic space for problems within  $P_c$ .

The oracle complexity of some specific problems in  $NP_c$  has recently been investigated. Rivest and Shamir [23] show that the oracle complexity of the language of composite numbers with natural evidence (i.e., factors) is at most  $n/3 + O(1)$ . Luks [18] has shown that graph isomorphism with natural evidence has oracle complexity  $O(\sqrt{n})$ . Using the canonical forms technique of Miller [19], the isomorphism problems for groups, Latin squares, and Steiner triple systems have oracle complexity  $O(\log^2 n)$ .

The following theorems summarize some properties of oracle complexity and demonstrate that our choice of oracle is optimal. We have restricted our purview to the three classes  $NP$ ,  $P$ , and  $LOGSPACE$ , as they are the most interesting from a practical standpoint and as they are also representative of the behavior of other complexity classes. The first two theorems have trivial proofs.

**Theorem 3.3.** (1) *If some  $NP_c$ -complete problem has logarithmic oracle complexity, then  $P = NP$ .*

(2) *If some  $P_c$ -complete problem has logarithmic oracle complexity, then  $P = LOGSPACE$ .*

(3) *If some  $NLOGSPACE_c$ -complete problem has logarithmic oracle complexity, then  $NLOGSPACE = LOGSPACE$ .*

**Theorem 3.4.** *If some  $NP_c$ -complete ( $P_c$ -complete,  $NLOGSPACE_c$ -complete) problem has polylogarithmic oracle complexity, then so do all problems in  $NP_c$  ( $P_c$ ,  $NLOGSPACE_c$ ).*

The next theorem indicates that the best possible oracle need never be outside the complexity class in which the problem sits.

**Theorem 3.5.** *If  $\Pi \in NP_c$  ( $P_c$ ,  $NLOGSPACE_c$ ) has oracle complexity less than or equal to  $f(n)$  with some fixed oracle  $A$ , then oracle complexity no greater than  $f(n)$  can be achieved for  $\Pi$  using an  $NP$ -complete ( $P$ -complete,  $NLOGSPACE$ -complete) oracle.*

**Proof.** Note that  $f(n)$  must be  $P$ -time constructible for  $\Pi \in NP_c$ , with similar conditions for the other two classes. We only sketch the proof. If the oracle set  $A$  sits within the class, but is not complete for it, we can simply reduce  $A$  to a complete set within the class and thus replace each query to  $A$  by an equivalent query to the complete set. If the set  $A$  does not sit within the class,  $A$  can be replaced by a set  $A'$  consisting of prefixes of those computation records of the oracle algorithm that produce acceptable evidence.  $\square$

We can pursue the consequences of this last result in terms of oracle choice. Our first corollary establishes that self-reduction is optimal for complete problems; its proof follows immediately from our last theorem.

**Corollary 3.6.** *If a problem is complete for one of these three classes, then it self-reduces as efficiently as it reduces to any oracle at all.*

Since it is easy to provide at least one self-reduction, it follows that complete problems for these three classes, in our constructive formalism, *always* self-reduce; such is not the case in the classical setting. Our second corollary shows that self-reduction can only be suboptimal for “incomplete” problems.

**Corollary 3.7.** (1) *If a problem in  $NP_c$  is found that self-reduces less efficiently than it does to a given oracle, then  $P \neq NP$ .*

(2) *If a problem in  $P_c$  is found that self-reduces less efficiently than it does to a given oracle, then  $P \neq LOGSPACE$ .*

(3) *If a problem in  $NLOGSPACE_c$  is found that self-reduces less efficiently than it does to a given oracle, then  $NLOGSPACE \neq LOGSPACE$ .*

Note that problems obeying the hypotheses cannot be complete (by our previous corollary) nor can they belong to a lower complexity class; hence they are incomplete problems (in the terminology of [3]).

Theorem 3.4 and Corollary 3.6 can be used as circumstantial evidence that a problem is not complete. For example, since group isomorphism has polylogarithmic oracle complexity, Theorem 3.4 suggests that it is highly unlikely that group isomorphism with natural evidence is  $NP_c$ -complete. Luks’ oracle algorithm for graph isomorphism, together with Corollary 3.6, provides evidence (further to that of [10]) that graph isomorphism is not NP-complete, since no one knows of a self-reduction using a sublinear number of queries.

Oracle complexity can be a useful tool in the design of algorithms. Consider the problem of determining whether a given graph has a vertex cover of size at most  $k$ , for any fixed  $k$ . The results of Robertson and Seymour immediately imply that there exists a quadratic-time algorithm for this problem. Using this (unknown) decision algorithm as an oracle, we can develop a cubic time search algorithm for the problem, which we then turn into a simple linear-time algorithm by eliminating the unknown decision oracle. Select any edge  $(u, v)$ ; delete  $u$  and, using the oracle, ask whether the remaining graph has a vertex cover of size at most  $k - 1$ . If so, put  $u$  in the vertex cover and recur; otherwise, put  $v$  in the vertex cover and recur. In  $k$  queries, a vertex cover has been generated when any exists. Now one can eliminate the oracle by trying all  $2^k$  possible sequences of responses; since  $2^k$  is a constant, the resulting algorithm runs in linear time for each value of  $k$ . Better yet, we *know* the algorithm!

#### 4. Blind oracles and reductions

Many natural self-reduction algorithms actually make no essential use of the in-

put [6]. Formalizing this observation provides another perspective on oracle algorithms and a way to measure their efficiency.

**Definition 4.1.** A blind oracle algorithm is an oracle algorithm which has only access to the length of the input, not the input itself; on a query to the oracle, the input is automatically prefixed to the query.

Thus a blind oracle algorithm attempting to produce evidence for string  $x$  only has access to the value  $|x|$ ; however, on query string  $y$ , the oracle actually decides membership for the string  $xy$ .

**Definition 4.2.** The blind oracle complexity of a problem  $\mathcal{R}$ , call it  $\text{boc}(\mathcal{R})$ , is the minimum number of oracle calls made to some fixed, but unrestricted language by an oracle algorithm (running within appropriate resource bounds) which uncovers acceptable evidence for the problem.

Information-theoretic arguments immediately give lower bounds on blind oracle complexity. For example, if  $\mathcal{R}$  is the equality relation, we clearly have  $\text{boc}(\mathcal{R}) = n$ . More interesting are bounds for some standard NP-complete problems.

**Theorem 4.3.** Let VC/Nat be the vertex cover problem with natural evidence (the cover) and HC/Nat the Hamiltonian circuit problem with natural evidence (the circuit).

$$\left\lceil \log \binom{n}{\lfloor n/2 \rfloor - 1} \right\rceil \leq \text{boc}(\text{VC/Nat}) \leq \left\lceil \log \binom{n}{\lfloor n/2 \rfloor} + \log n \right\rceil,$$

$$\left\lceil \log \left( 1 + \frac{n!}{2n} \right) \right\rceil \leq \text{boc}(\text{HC/Nat}) \leq (n-1) \lceil \log(n-1) \rceil.$$

Hence  $\text{boc}(\text{VC/Nat}) \in \Theta(n)$  and  $\text{boc}(\text{HC/Nat}) \in \Theta(n \log n)$ .

**Proof.** The upper bounds are derived from simple oracle algorithms for each problem (that for VC actually finds a minimal cover for the problem). The lower bounds come from simple counts of the number of distinct possible arrangements that may have to be checked to identify a solution.  $\square$

Is blind oracle complexity preserved in some sense through reductions? The main problem here is that our reductions are not themselves blind and so defeat the blindness of the oracle algorithms by giving them a description of the input as a side-effect. We need a type of reduction which preserves blindness (such is theory!). Such a reduction must perforce use a very different mechanism from that of constructive many-one reductions. Let us use an anthropomorphic analogy. In the latter style of reduction, the scientist with the new problem calls upon a scientist with a known

complete problem, who then serves as a one-shot oracle: the first communicates to the second the transformed instance and the second returns to the first suitable evidence for the transformed instance. The second scientist acts in mysterious ways (i.e., unknown to the first) and thus has attributes of deity. But in a blind reduction, neither scientist is allowed to see the input and yet the instance given the first scientist must be transformed into the instance given the second; both scientists then sit at the same level, as humble supplicants to some all-powerful deity. The reduction goes as follows: the first scientist asks the oracle to carry out the transformation  $x \rightarrow f(x)$  implicitly (neither  $x$  nor  $f(x)$  will be made known), then uses the oracle algorithm provided by the second scientist to establish a certificate  $y'$  for the unknown  $f(x)$ , and finally applies  $g$  to recover evidence  $y$  for instance  $x$  from the known values of  $y'$  and  $|x|$ . Since the oracle algorithm of the second scientist assumes knowledge of the instance size, in this case  $|f(x)|$ , it is imperative that  $|f(x)|$  be computable from  $|x|$ ; hence a blind reduction must be uniform with respect to instance sizes. Note that what gets communicated in the blind reduction is the oracle protocol, whereas what gets communicated in the normal many-one reduction is the (transformed) instance.

**Definition 4.4.** A blind (constructive) many-one reduction is a many-one constructive reduction,  $(f, g)$ , where the first map,  $f$ , is length-uniform (i.e.,  $|f(x)|$  depends only on  $|x|$ ) and the second map,  $g$ , only has access to the size of the original input.

Now we can check that blind reductions indeed preserve blind oracle complexity; we state this only for the case of most interest to us.

**Lemma 4.5.** *If  $\mathcal{R}_1 \in \text{NP}_c$  blindly reduces to  $\mathcal{R}_2 \in \text{NP}_c$  and  $\mathcal{R}_2$  has polylogarithmic oracle complexity, so does  $\mathcal{R}_1$ .*

The proof is obvious from our anthropomorphic discussion above.

With a blind transformation, we can define blind completeness in the obvious way. Perhaps surprisingly, blindness does not appear to affect the power of reductions very much, as the following claim shows.

**Claim 4.6.** *All known NP-complete sets are the domains of blindly  $\text{NP}_c$ -complete relations.*

Obviously, we cannot offer a proof of this statement; we simply remark that all known reductions between NP-complete problems can easily be made length-uniform and that, in all of these reductions, the evidence assumes such characteristic form that, armed only with the length of the original instance and evidence for the transformed instance, we can easily reconstruct evidence for the original instance. (This obviously is strongly reminiscent of the observation that all known reductions among NP-complete problems can be made weakly parsimonious, so that the

number of different certificates for one problem can easily be recovered from the number of different certificates for the transformed instance.)

A fundamental question in classical complexity theory concerns the density of sets in various classes (recall that the density of a set is the rate of growth of its membership as a function of the length of the elements). A set  $S$  is *sparse* if  $|\{x \mid x \in S, |x| = n\}| \leq p(|x|)$  for some polynomial  $p$ ; it has *subexponential* density (or is *semi-sparse*) if  $|\{x \mid x \in S, |x| = n\}| \in O(2^{\log^k n})$ , for some positive integer  $k$ . It is widely suspected that NP-complete sets must have exponential density; however, all that is known at this time is that NP-complete sets cannot be sparse unless  $P = NP$  (even their complexity cores cannot at present be shown to have more than subexponential density [21]). Blind oracles allow us to prove in our context a much stronger result about density.

**Theorem 4.7.** *There is no blindly  $NP_c$ -complete relation with domain of subexponential density.*

**Proof.** We have shown that VC/Nat has linear blind oracle complexity, which is not a polylogarithmic function. Hence the domain of VC/Nat has density greater than subexponential. Since polylogarithmic oracle complexity is preserved through blind reductions, it follows that no other  $NP_c$ -complete problem can have a domain of subexponential density.  $\square$

Combining this result with a proof for our claim would allow us to transfer our conclusion to NP-complete sets; it might also help in characterizing the relationship between the sets NP and POLYLOGSPACE.

## 5. Conclusions

We have presented a proposal for a constructive theory of complexity. By examining reducibility and oracle algorithms, we have been able to establish a number of simple results which show that our theory has a sound basis and holds much promise. Indeed, through the use of blind oracle methods, we have been able to prove within our framework a much stronger result than has been shown to date in the classical theory.

Much work obviously remains to be done. Problems of particular interest to us at this time include a further study of the relationship between decision, checking, and search. For instance, Schnorr [27] conjectures that there exist P-decidable predicates that are not P-searchable if we require particularly concise evidence; this is the type of question that may be advantageously addressed within our framework. Another problem of special interest is the characterization of blindly complete relations in a variety of classes and the connections between blind reductions and communication complexity. Of potential interest is a study of the higher classes of

complexity (PSPACE, EXP); although these classes can hardly be deemed constructive from a practical standpoint (any relation that is not P-checkable is only "solvable" in some abstract sense), the greater resources which they make available may enable us to derive some interesting results with respect to reducibility.

### Acknowledgement

We would like to thank Len Adleman, Eric Allender, Juris Hartmanis, Richard Karp, Gene Luks, Steve Mahaney, and Doug Stinson for helpful conversations and encouragement.

### References

- [1] J.L. Balcazar, Self-reducibility, in: Proceedings Symposium on Theoretical Aspects of Computer Science STACS-87 (1987) 136-147.
- [2] M. Beeson, Foundations of Constructive Mathematics (Springer, Berlin, 1980).
- [3] G. Brassard and C. Crepeau, Nontransitive transfer of confidence: A perfect zero-knowledge interactive protocol for SAT and beyond, in: Proceedings 27th Symposium on Foundations of Computer Science FOCS-86 (1986) 188-195.
- [4] M.R. Fellows and M.A. Langston, Nonconstructive advances in polynomial-time complexity, Inform. Process. Lett. 26 (1987) 157-162.
- [5] M.R. Fellows and M.A. Langston, Nonconstructive tools for proving polynomial-time decidability, J. ACM 35 (1988) 727-739.
- [6] M.R. Fellows and M.A. Langston, Fast self-reduction algorithms for combinatorial problems of VLSI design, in: Proceedings 3rd Aegean Workshop on Computing (1988) 278-287.
- [7] M.R. Fellows and M.A. Langston, On search, decision, and the efficiency of polynomial-time algorithms, in: Proceedings 21st Annual ACM Symposium on Theory of Computing STOC-89 (1989) 501-512.
- [8] H. Friedman, N. Robertson and P.D. Seymour, The metamathematics of the graph minor theorem, in: Applications of Logic to Combinatorics (Amer. Math. Soc., Providence, RI, to appear).
- [9] M.R. Garey and D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness (Freeman, San Francisco, CA, 1979).
- [10] O. Goldreich, S. Micali and A. Wigderson, Proofs that yield nothing but their validity and a methodology of cryptographic protocol design, in: Proceedings 27th Symposium on Foundations of Computer Science FOCS-86 (1986) 174-187.
- [11] R. Impagliazzo and E. Tardos, Private communication reported by E. Allender.
- [12] D.S. Johnson, The many faces of polynomial time, in: The NP-Completeness Column: An Ongoing Guide, J. Algorithms 8 (1987) 285-303.
- [13] R.M. Karp, E. Upfal and A. Wigderson, Are search and decision problems computationally equivalent, in: Proceedings 17th ACM Symposium on Theory of Computing STOC-85 (1985) 464-475.
- [14] Ker-I Ko, On helping by robust oracle machines, Theoret. Comput. Sci. 52 (1987) 15-36.
- [15] M. Krentel, The complexity of optimization problems, Ph.D. Dissertation, Department of Computer Science, Cornell University, Ithaca, NY (1987).
- [16] R.E. Ladner, The circuit value problem is log-space complete for  $\mathcal{P}$ , SIGACT News 7 (1975) 18-20.

- [17] L.A. Levin, Universal sequential search problems, *Problemy Peredachi Informatsii* 9 (1973) 115–116 (in Russian).
- [18] E.M. Luks, Private communication.
- [19] G.L. Miller, On the  $n^{\log n}$  isomorphism technique, in: *Proceedings 10th ACM Symposium on Theory of Computing STOC-78* (1978) 51–58.
- [20] J. Nešetřil and R. Thomas, Well quasi orderings, long games and a combinatorial study of undecidability, in: *Contemporary Mathematics* 65 (Amer. Math. Soc., Providence, RI, 1987) 281–293.
- [21] P. Orponen and U. Schöning, The density and complexity of polynomial cores for intractable sets, *Inform. and Control* 70 (1986) 54–68.
- [22] M.O. Rabin, Probabilistic algorithms for testing primality, *J. Number Theory* 12 (1980) 128–138.
- [23] R.L. Rivest and A. Shamir, Efficient factoring based on partial information, in: *Eurocrypt 85, Lecture Notes in Computer Science* 219 (Springer, Berlin, 1985) 31–34.
- [24] N. Robertson and P. Seymour, Disjoint paths—a survey, *SIAM J. Algebraic Discrete Methods* 6 (1985) 300–305.
- [25] N. Robertson and P. Seymour, Graph minors—a survey, in: J. Anderson, ed., *Surveys in Combinatorics* (Cambridge Univ. Press, Cambridge, 1985) 153–171.
- [26] W.J. Savitch, Nondeterministic  $\log n$  space, in: *Proceedings 8th Annual Princeton Conference on Information Sciences and Systems* (1974) 21–23.
- [27] C. Schnorr, On self-transformable combinatorial problems, *Math. Programming Stud.* 14 (1981) 225–243.
- [28] U. Schöning, Robust algorithms: A different approach to oracles, *Theoret. Comput. Sci.* 40 (1985) 57–66.
- [29] L. Valiant, Relative complexity of checking and evaluating, Report, University of Leeds, Leeds (1974).

## Fast Search Algorithms for Layout Permutation Problems

MICHAEL R. FELLOWS  
*University of Idaho*

MICHAEL A. LANGSTON  
*University of Tennessee and Washington State University*

We have previously established the existence of *decision* algorithms with low-degree polynomial running times for a number of difficult combinatorial problems, including many that can be stated in terms of VLSI layout, placement, embedding, and routing. In this paper, we turn our attention to the *search* complexity of these problems. We introduce a general technique, which we term *scaffolding*, and illustrate how it is useful in the design of efficient search algorithms.

---

CAD tools, layout algorithms, search complexity, VLSI design paradigms

---

### 1 INTRODUCTION

Mathematical tools are now available with which the existence of asymptotically fast decision algorithms can be proved nonconstructively [1-7] (a brief exposition is contained in the Appendix). In a recent series of papers [8-11], we have employed these new tools to prove low-degree polynomial-time decision complexity for a variety of combinatorial problems. Included on this list are a number of fixed-parameter layout permutation problems, many of which are well known for their relevance to VLSI design. Some of these problems were not previously known to be decidable in polynomial time at all; others were known to be decidable only in polynomial time by way of brute force or dynamic programming algorithms with unboundedly high-degree polynomial running times (i.e., the degree of the polynomial is an unbounded function of the relevant parameter).

For our purposes, it is important to note that, in general, the algorithms proven to exist with these new tools decide only whether a "yes" or a "no" instance has

---

A preliminary version of a portion of this paper was presented at the Third Aegean Workshop on Computing held on Corfu Island, Greece, in June 1988. This research has been supported in part by the National Science Foundation under Grants MIP-8603879 and MIP-8919312, by the Office of Naval Research under Contracts N00014-88-K-0343 and N00014-88-K-0456, and by NASA under Engineering Research Center Grant NAGW-1406.

A portion of Michael A. Langston's work was performed while visiting the Institute for Mathematics and Its Applications at the University of Minnesota, Minneapolis, MN, and while visiting the Coordinated Science Laboratory at the University of Illinois, Urbana, IL.

Correspondence and requests for reprints should be sent to Michael A. Langston, Department of Computer Science, University of Tennessee, Knoxville, TN 37996-1301.

■ Manuscript received June 1, 1989; manuscript revised December 1, 1989.

been presented. That is, unlike algorithms devised with traditional methods, algorithms based on these tools typically rely on the existence of finite lists of "negative" evidence (obstruction sets) rather than on attempts to find more natural "positive" evidence (satisfactory layouts).

In this paper, we explore the complexity of *search* problems (where the goal is to produce positive evidence when it exists [12]) that correspond to the decision problems suggested earlier. Given a decision problem  $\Pi_D$  and the corresponding search problem  $\Pi_S$ , any method that isolates a solution to  $\Pi_S$  by repeated calls to an algorithm that answers  $\Pi_D$  is commonly termed a *self-reduction*. Although we have previously identified straightforward self-reduction methods that can be applied to many layout permutation problems [13, 11], we herein develop considerably more efficient search strategies. To accomplish this, we introduce a general technique that we call *scaffolding*, and show how it can be the basis for fast search algorithms for a number of layout permutation problems, including MIN CUT LINEAR ARRANGEMENT, GATE MATRIX LAYOUT, and several others. Because  $O(n^2)$  time decision algorithms exist for each problem we consider [10, 11], and because we use at most  $O(n \log n)$  calls to each decision algorithm, the search algorithms we devise require only  $O(n^3 \log n)$  time.

In the next section, we discuss the self-reduction process and define a few useful terms. In Section 3, we outline the main features of scaffolding. In the final section, we present the problem-specific implementation details necessary to apply scaffolding to a number of illustrative problems.

## 2 COMPLEXITY OF SEARCHING

A natural computational setting in which to consider the relationship between search and decision problem complexities is that of oracle computations, where an *oracle algorithm*  $A$  to solve a search problem has access to some decision problem oracle  $O$ . More formally, an oracle algorithm is modeled as a random access machine with a special query register and three special states,  $s_{\text{query}}$ ,  $s_{\text{yes}}$ , and  $s_{\text{no}}$ . On entering the state  $s_{\text{query}}$ , the machine makes a transition in unit time either to state  $s_{\text{yes}}$  or to state  $s_{\text{no}}$ , depending on whether the string  $q.i$ , where  $q$  denotes the contents of the query register and  $i$  denotes the input, belongs to the oracle language.

The *overhead* of  $A$  is the time required by  $A$  to pinpoint a solution, if any exist, where each invocation of  $O$  is charged only a unit-time cost. Thus, the overhead of  $A$  is the time required by  $A$  outside of the running time of the oracle. Since our goal is to devise  $O(n^3 \log n)$  oracle algorithms, it is necessary to ensure that none requires more than  $O(n^3 \log n)$  overhead. [In fact, none will need more than  $O(n^2)$  overhead.]

For an example, consider the vertex permutation problem known as MIN CUT LINEAR ARRANGEMENT [12]. In this  $\mathcal{NP}$ -complete problem, we are given a graph  $G$  and a positive integer  $k$ , and are asked whether  $G$  can be laid out with its vertices along a straight line so that no orthogonal cutting plane that intersects the line between any two consecutive vertices (but not on a vertex) ever cuts more than  $k$  edges. Until recently, the fastest known algorithm for both the decision and

the search versions of this problem was a dynamic programming formulation with time complexity  $O(n^{k-1})$  [14], where  $n$  denotes the number of vertices in  $G$ . Thus, MIN CUT LINEAR ARRANGEMENT is in  $\mathcal{P}$  for any fixed value of  $k$ . We have shown [11], however, that the asymptotic time complexity can be reduced to  $O(n^2)$ , for any fixed  $k$ . This can be used to obtain an  $O(n^4)$  search strategy.

#### Theorem 1 [13]

For any fixed  $k$ , a satisfactory solution to MIN CUT LINEAR ARRANGEMENT can be constructed, if any exist, by an oracle algorithm [with overhead  $O(n^2)$ ] that makes  $O(n^2)$  calls to an  $O(n^2)$  decision oracle for MIN CUT LINEAR ARRANGEMENT.

If, as in Theorem 1, the oracle language consulted by the oracle algorithm solving the search problem is precisely the set of "yes" instances for the corresponding decision problem, then this is termed a self-reduction. A novelty of our approach is that we shall, in the sequel, obtain efficient oracle algorithms using oracle languages for closely related, but different decision problems, with no increase in the degree of the polynomial bounding the asymptotic time complexity of the decision oracle.

How good is Theorem 1? More specifically, can we do better with a more sophisticated self-reduction strategy? Clearly, there may be trade-offs between the amount of overhead time required, the number of oracle calls issued, and the power (computational cost) of the oracle used.

The goal of the next two sections is to show that, for MIN CUT LINEAR ARRANGEMENT and related problems, there are oracle languages that are both recognizable in quadratic time and yet powerful enough to require only  $O(n \log n)$  calls by an oracle algorithm [with  $O(n^2)$  overhead]. This naturally yields the desired  $O(n^3 \log n)$  time search algorithms.

From a practical standpoint, these algorithms are of increasing interest as the large constants and nonconstructive nature of the  $O(n^2)$  oracles are reduced or eliminated [15–17]. From a more purely theoretical perspective, it can be shown that these methods are "blind," needing access to the input only indirectly through queries to the oracle, and "fast," running to within a constant factor as fast as any blind strategy can. (In fact, if one is willing to consider nonblind schemes, even faster self-reductions are sometimes possible [18, 15].)

As is usual, we intend that  $n$  be reserved to denote the length of the input. For the class of layout permutation problems we focus on, however, we shall abuse this notation slightly, using  $n$  to denote the number of vertices in the input graph. (This causes no problem, because the family of "yes" instances for each individual problem permits only a linear number of edges. This bound follows from a result originally derived in [19]. See [11, 6] for details.)

### 3 SCAFFOLDING—AN OVERVIEW

We now describe a general method for the design of oracle algorithms that use oracle languages closely related to the relevant decision problems and are, there-

fore, recognizable in low-degree polynomial time. This method is particularly applicable to layout permutation problems concerning *width* metrics on graphs and hypergraphs, where the metric is defined to be the minimum, over all permutations of the vertex or edge set, of an objective function defined on such permutations. These objective functions arise in a number of  $\mathcal{NP}$ -complete problems, including MIN CUT LINEAR ARRANGEMENT, MODIFIED MIN CUT [20], PATH WIDTH [2], GATE MATRIX LAYOUT [21], VERTEX SEPARATION [22], SEARCH NUMBER [23], NODE SEARCH NUMBER [24], TWO-DIMENSIONAL GRID LOAD FACTOR [10], and others. Each of these is known to possess an  $O(n^2)$  time decision algorithm for any fixed width value [10, 11].

Our strategy yields efficient search algorithms for problems that satisfy the following *uniformity condition* concerning the complexity of the associated fixed-parameter decision problem with the width metric  $w$ : there is a constant  $c$  such that, for *all*  $k$ , it can be decided in time  $O(n^c)$  for an arbitrary graph  $G$  whether  $w(G)$  is at most  $k$ . The oracle algorithms we shall describe for width- $k$  layout problems exploit this uniformity condition by employing oracle languages that we can show are efficiently reducible to the width- $k'$  decision problem, for an appropriately chosen  $k'$  generally greater than  $k$ . (Of course, we must ensure that both the oracle queries and the width- $k'$  problem instances we generate in this manner have size linear in  $n$ .)

Our method proceeds in two stages. First, we describe a convenient oracle language  $L$  that supports an efficient oracle algorithm, solving the width- $k$  search problem by imitating the well-known binary insertion sort algorithm [25]. Next we show that  $L$  can be recognized in low-degree polynomial time by reducing the problem of recognizing  $L$  to the decision problem for width- $k'$  layout.

The reduction consists primarily of attaching to  $G$  a *scaffolding component* that encodes the data structure for the sorting algorithm (i.e., we use  $L$  to describe valid intermediate configurations of the data structure). In the case of a vertex permutation problem, for example, the vertices of  $G$  are attached one by one to the *template level* of the scaffold as a permissible sorted order of the attached vertices is progressively extended. [Such a permissible order on a subset of  $V(G)$  is one that can be extended to a permutation of  $V(G)$  with width  $k$  or less.] The

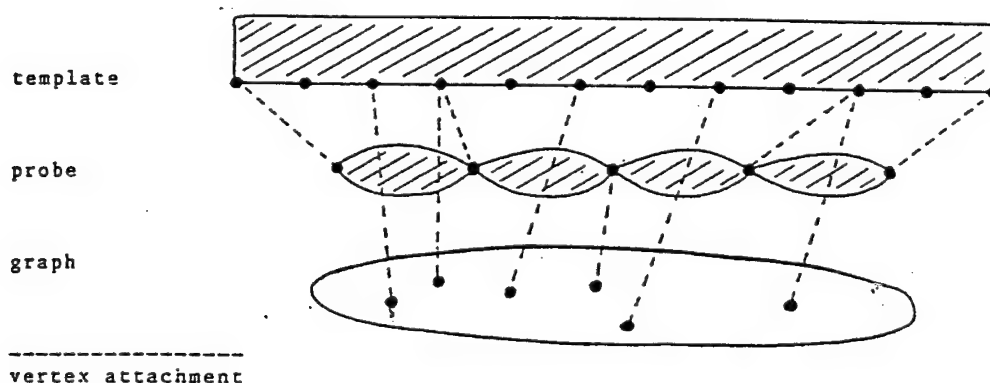


Figure 1. Overview of the scaffolding technique.

determination of where, among the already attached vertices, a newly chosen unattached vertex may be inserted in a permissible order is made by means of the *probe level* of the scaffold. Attachments at this level encode a determination of whether a vertex  $v$  can be inserted in any permissible order between (not necessarily consecutive) vertices  $x$  and  $w$  that are already attached to the template. The overall idea is to use the scaffold to help "rigidify" the graph in an efficient manner. An overview of this general construction is depicted in Figure 1. Details for a few illustrative problems will be presented in the next section.

#### 4 ON THE IMPLEMENTATION OF SCAFFOLDING

The choice of  $k'$  and the details of the scaffold component are specific to the particular layout problem considered. Attachments can take the form of edge additions, vertex identifications, or more complex graph gadgets. We continue our presentation with the archetypical problem MIN CUT LINEAR ARRANGEMENT.

##### Theorem 2

For any fixed  $k$ , a satisfactory solution to MIN CUT LINEAR ARRANGEMENT can be constructed, if any exist, by an oracle algorithm [with overhead  $O(n^2)$ ] that makes  $O(n \log n)$  calls to an  $O(n^2)$  decision oracle.

*Proof.* For convenience, we shall henceforth use the term *cutwidth* to denote the metric of relevance to MIN CUT LINEAR ARRANGEMENT. That is, the cutwidth of a layout of a graph  $G$  is the maximum number of edges cut by any plane orthogonal to the layout; the cutwidth of  $G$  is the least value of  $k$  for which  $G$  has a layout with cutwidth  $k$ . In addition, we make the reasonable assumption concerning our encoding scheme that it allows for queries that unambiguously name the vertices of the input graph.

Consider the following decision problem, which will correspond to our desired oracle language  $L$  in a way that we shall make precise shortly.

*Input:* A sextuple  $(G, V', \leq, x, v, w)$ , where  $G = (V, E)$  is a graph.  $V' \subseteq V$ ,  $\leq$  is a linear order of  $V'$ , and  $\{x, v, w\}$  is a set of distinct elements of  $V$  with  $\{x, w\} \subseteq V'$ .

*Question:* Is there a linear order on  $V$ , corresponding to a layout of  $G$  with cutwidth at most  $k$ , that extends  $\leq$  such that  $x \leq v \leq w$ ?

It is easy to see that an oracle algorithm can construct a linear order on  $V$  that corresponds to a satisfactory layout of  $G$ , when any exist, by making  $O(n \log n)$  calls to this decision problem, performing a binary insertion sort on  $V$ . Specifically, we start with  $V'$  containing two arbitrarily chosen vertices of  $G$ , and progressively insert each vertex of  $V - V'$  by binary search until a complete layout is determined. This approach can be implemented using the oracle language  $L = \{q \cdot G \mid q = (V', \leq, x, v, w) \text{ and } (G, V', \leq, x, v, w) \text{ is a "yes" instance to the decision problem}\}$ .

To show that  $L$  is recognized in  $O(n^2)$  time, we describe a reduction to the problem of recognizing graphs of cutwidth at most  $3k$ . (We choose the value  $3k$  solely because it is an obvious candidate when augmenting  $G$  with template and probe levels. Clearly,

larger values can be used as well.) It suffices to restrict our attention to input  $q.G$ ,  $q = (V', \leq, x, v, w)$ , in which  $V' \cup \{v\}$  meets every component of  $G$ . (Disjoint components can be laid out separately.)

In quadratic time we will construct a graph  $G_q$  that has cutwidth at most  $3k$  if, and only if,  $q.G \in L$ . Before presenting the details of this construction, we use Figure 2 to depict the scaffolding of an arbitrary graph as it might appear during some step of the algorithm, with  $k = 2$ . (We shall work from left to right, ensuring that if, at any step, the binary insertion of  $v$  fails for all of  $V'$ , then it must be that  $v$  can be placed to the right of all of  $V'$ .)

Let  $T$  denote the *template* level of  $G_q$ . Its vertex set is  $\{t_0, t_1, \dots, t_{n+1}\}$ . For  $i = 1, 2, \dots, n-1$ , a set of  $k$  (multiple) edges joins  $t_i$  with  $t_{i+1}$ . A set of  $3k$  edges joins  $t_0$  to  $t_1$ , and a set of  $3k$  edges joins  $t_n$  to  $t_{n+1}$ .

Let  $P$  denote the *probe* level of  $G_q$ . The vertex set of  $P$  is  $\{p_1, p_2, \dots, p_3\}$  and, for  $i = 1, 2, \dots, 4$ , a set of  $k$  edges joins  $p_i$  to  $p_{i+1}$ . The graph  $G_q$  is obtained from  $T$ ,  $P$ , and  $G$  by the following sequence of vertex identifications:

1. The vertices  $t_1, t_2, \dots, t_{|V'|}$  are identified one-to-one with the vertices of  $V'$  in the order specified by  $\leq$ .
2. The vertex  $t_1$  is identified with the vertex  $p_1$ , and the vertex  $t_{|V'|}$  is identified with the vertex  $p_3$ .
3. The vertex  $p_2$  is identified with the vertex  $x$ , and the vertex  $p_4$  is identified with the vertex  $w$ .
4. The vertex  $p_3$  is identified with the vertex  $v$ .

Note that since each component of  $G$  meets  $V' \cup \{v\}$ , the resulting graph  $G_q$  is connected. It remains to argue that  $G_q$  has cutwidth at most  $3k$  if, and only if,  $q.G \in L$ .

Suppose that  $\leq$  can be extended to a linear order on  $V$  that witnesses  $q.G \in L$ . Let  $\sigma_T = (t_0, t_1, \dots, t_{n+1})$  be the sequence of vertices of  $T$ . Let  $\sigma_P = (p_1, p_2, \dots, p_3)$  be the sequence of vertices of  $P$ . The linear order  $\leq$  on  $V$  can be represented by a sequence of vertices  $\sigma_G = (v_{i_1}, v_{i_2}, \dots, v_{i_n})$ . The graphs  $T' = T - \{t_0, t_{n+1}\}$ ,  $P$ , and  $G$  each have cutwidth at most  $k$  with layouts described, respectively, by the sequences  $\sigma_T = (t_1, t_2, \dots, t_n)$ ,  $\sigma_P$ , and  $\sigma_G$ . It follows that if  $\sigma$  represents any layout for  $G_q$  that respects  $\sigma_T$ ,  $\sigma_P$ , and  $\sigma_G$  and that has an initial subsequence specified by  $(t_0, t_1)$  and a final subsequence specified by  $(t_n, t_{n+1})$ , then  $\sigma$  describes a layout of width of at most  $3k$ . Since the given order on  $V'$  extends to the order represented by  $\sigma_G$  in a way that achieves  $x \leq v \leq w$ , we may choose  $\sigma$  so that the vertices identified in the construction

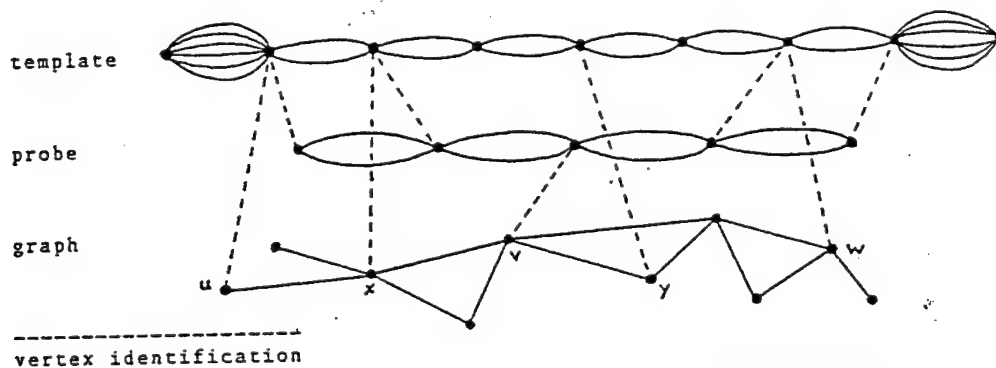


Figure 2. Sample scaffold for MIN CUT LINEAR ARRANGEMENT.

of  $G_q$  from  $T$ ,  $P$ , and  $G$  appear consecutively in  $\sigma$ , and, therefore, the cutwidth is unaffected by these identifications. Thus,  $G_q$  has cutwidth at most  $3k$ .

Conversely, suppose  $G_q$  has cutwidth bounded by  $3k$  as witnessed by a vertex solution sequence  $\sigma$ . Since  $G_q$  is connected, and since there are  $3k$  edges between  $t_0$  and  $t_1$  and between  $t_n$  and  $t_{n+1}$ ,  $\sigma$  must have its initial and final subsequences specified by  $(t_0, t_1, \dots, t_n, t_{n+1})$  or the reverse of this, which we can ignore without loss of generality. Observe that the restriction of  $\sigma$  to the vertex set of  $T$  must equal  $\sigma_T$  (in other words,  $\sigma$  cannot describe a layout in which  $T$  is "kinked"), else the cutwidth of the layout described by  $\sigma$  exceeds  $3k$ , which is impossible. It follows that the restriction of  $\sigma$  to the vertices of  $V'$  respects the given order  $\leq$ , according to our construction of  $G_q$ . Similarly, the restriction of  $\sigma$  to the vertex set of  $P$  must equal  $\sigma_P$ , and, therefore, according to the identifications made in the construction of  $G_q$ , the vertex  $x$  must precede the vertex  $v$  in  $\sigma$ , and  $v$  must precede  $w$ . Since there are a total of  $2k$  edges from the levels  $T$  and  $P$  to be cut between any two positions between  $t_1$  and  $t_n$  in the layout described by  $\sigma$ , and since the cutwidth  $G_q$  is bounded by  $3k$ , it follows that  $\sigma$  describes a layout of  $G$  with cutwidth at most  $k$ . Therefore  $q \cdot G \in L$ .  $\square$

MIN CUT LINEAR ARRANGEMENT has been a useful example of a vertex permutation problem amenable to this approach. We now turn our attention to GATE MATRIX LAYOUT, an  $\mathcal{NP}$ -complete problem that was originally posed in terms of operations on Boolean matrices. Formally, we are given an  $n \times m$  Boolean matrix  $M$  and an integer  $k$ , and are asked whether we can permute the columns of  $M$  so that, if in each row we change to asterisks every zero lying between the row's leftmost and rightmost 1, then no column contains more than  $k$  1s and asterisks. We refer the interested reader to [21] for sample instances, figures, and additional background on this challenging combinatorial problem.

We have shown that GATE MATRIX LAYOUT is linear-time equivalent to a natural edge permutation problem on graphs [8], and we have used this to show that, for any fixed  $k$ , the decision problem is solvable in  $O(n^2)$  time. The equivalent edge permutation problem is described as follows. As before,  $n$  denotes the number of vertices in a graph. Let  $E_v$  denote the set of edges incident on a vertex  $v$ . For a bijection  $f$  from  $E$  to  $\{1, 2, \dots, |E|\}$ , the *span* of  $v$  under  $f$  is the set of consecutive integers  $\{i, i+1, \dots, j\}$  for which  $i$  is the least element in  $f(E_v)$  and  $j$  is the greatest. In the decision problem that we shall henceforth term EDGE PERMUTATION WIDTH, we are given a graph  $G$  and a positive integer  $k$ , and are asked whether there exists such a bijection with a property that, for each edge  $e$ ,  $f(e)$  is contained in the spans of at most  $k$  distinct vertices.

It is interesting that, for the purpose of providing polynomial-time decidability, it is advantageous to reduce GATE MATRIX LAYOUT to EDGE PERMUTATION WIDTH (as described in [8]) and work with graphs, while, for the purpose of devising efficient search strategies, it is more useful to reduce EDGE PERMUTATION WIDTH to GATE MATRIX LAYOUT (as we shall do with our scaffolding construction) and work with Boolean matrices.

We wish to state our results in terms of  $n$ , the number of vertices in the input graph. As we have previously pointed out, there is a linear bound on the number of distinct edges of any "yes" instance, so that  $n$  is a reasonable measure of the size of the input. (This is complicated only trivially by the possible existence of

loops and duplicate edges, which have no effect on this width metric. Therefore, we make the assumption that the input is restricted to simple graphs.)

### Theorem 3

For any fixed  $k$ , a satisfactory solution to EDGE PERMUTATION WIDTH can be constructed, if any exist, by an oracle algorithm [with overhead  $O(n^2)$ ] that makes  $O(n \log n)$  calls to an  $O(n^2)$  decision oracle.

*Proof.* For convenience, we shall henceforth use the term *cost* to denote the metric of relevance to EDGE PERMUTATION WIDTH and GATE MATRIX LAYOUT. That is, the cost of an edge permutation of a graph  $G$  is the maximum number of vertices whose spans contain a common edge's image under the permutation mapping; the cost of a column permutation of a Boolean matrix  $M$  is the maximum number 1s and asterisks in any column.

As in the proof of Theorem 2, we first define an oracle language  $L$  that supports a satisfactory oracle algorithm for the problem. The decision problem that corresponds to  $L$  is defined as follows.

*Input:* A sextuple  $(G, E', \leq, e_0, e_1, e_2)$ , where  $G = (V, E)$  is a graph,  $E' \subseteq E$ ,  $\leq$  is a linear ordering of  $E'$ , and  $\{e_0, e_1, e_2\}$  is a set of distinct elements of  $E$  with  $\{e_0, e_2\} \subseteq E'$ .

*Question:* Is there a linear order on  $E$  corresponding to an edge permutation of  $G$  with cost at most  $k$  that extends  $\leq$  such that  $e_0 \leq e_1 \leq e_2$ ?

Let  $L = \{q.G \mid q = (E', \leq, e_0, e_1, e_2) \text{ and } (G, E', \leq, e_0, e_1, e_2) \text{ is a "yes" instance to the decision problem}\}$ . There is a straightforward oracle algorithm with oracle language  $L$  that performs a binary insertion sort to construct a satisfactory edge permutation, when any exist, by making  $O(n \log n)$  oracle calls and requiring  $O(n^2)$  overhead.

To show that  $L$  can be recognized in  $O(n^2)$  time, we shall describe a scaffolding reduction to GATE MATRIX LAYOUT, which is decidable in  $O(n^2)$  time for each fixed value of the parameter. To accomplish this, we construct a matrix  $M(G, q)$  that has a cost of at most  $5k$  if and only if  $q.G \in L$ . (The value  $5k$  is chosen solely for simplicity.)

Before presenting the details of this reduction, we use Figure 3 to depict the scaffolding matrix  $M(G, q)$  corresponding to a sample string  $q.G$ . In this sample,  $k = 2$ ,  $q = (\{2, 3, 7\}, (2, 3, 7), 3, 1, 7)$ , and  $G = (V, E)$  with  $V = \{a, b, c, d, e\}$  and  $E = \{(a, b), (b, c), (a, c), (a, d), (c, d), (a, e), (d, e)\}$ .

The vertices of  $G$  are represented by a subset of the rows of the matrix, and the edges of  $G$  are represented by columns of the matrix containing exactly two 1s in that subset (but other 1s elsewhere). In arguing that  $L$  can be recognized in  $O(n^2)$  time, it suffices to restrict our attention to input  $q.G$ ,  $q = (E', \leq, e_0, e_1, e_2)$ , for which  $E' \cup \{e_1\}$  meets every component of  $G$ .

To specify formally the construction of  $M(G, q)$  from  $q.G$ , let  $R = \{r_i \mid 1 \leq i \leq t\}$  denote the set of rows of  $M(G, q)$ .  $M(G, q)$  will contain  $|E| = O(n)$  columns, each denoted by a subset  $c \subseteq R$ , with the understanding that  $c$  is the set of rows in which that column contains a 1 and that in all other rows that column contains a 0. The exact value of the upper bound  $t$  used in the indexing of  $R$  is implicit in the construction (and is not important, except that it must be a linear function of  $n$ ).

The *template level*  $T$  of  $M(G, q)$  is the set of columns:

1.  $t_0 = \{r_1, \dots, r_{3k}\}$ .
2. For  $i = 1, 2, \dots, |E'|$ ,  $t_i = \{r_{(i+1)k+1}, \dots, r_{(i+3)k}\}$ .
3.  $t_{|E'|+1} = \{r_{(|E'|+2)k+1}, \dots, r_{(|E'|+5)k}\}$ .

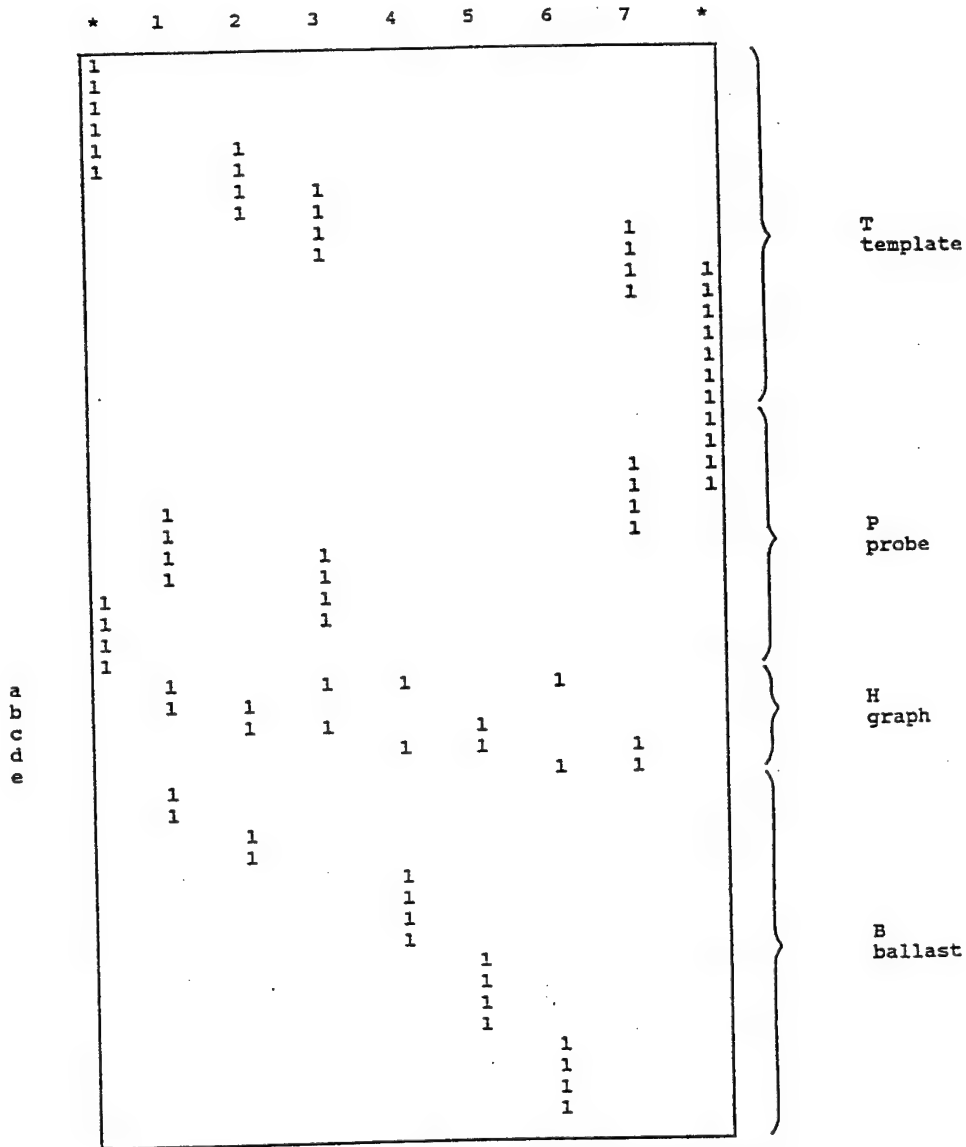


Figure 3. Sample scaffold for GATE MATRIX LAYOUT.

The probe level  $P$  of  $M(G, q)$  is the set of columns:

1.  $p_1 = \{r_{(|E|+5)k+1}, \dots, r_{(|E|+7)k}\}.$
2.  $p_2 = \{r_{(|E|+6)k+1}, \dots, r_{(|E|+8)k}\}.$
3.  $p_3 = \{r_{(|E|+7)k+1}, \dots, r_{(|E|+9)k}\}.$
4.  $p_4 = \{r_{(|E|+8)k+1}, \dots, r_{(|E|+10)k}\}.$
5.  $p_5 = \{r_{(|E|+9)k+1}, \dots, r_{(|E|+11)k}\}.$

The graph level  $H$  of  $M(G, q)$  contains, for each edge  $e = uv \in E$ , the column  $H_e = \{r_{h(u)}, r_{h(v)}\}$ , where  $h$  is a one-to-one map from  $V$  to  $\{r_{(|E|+11)k+1}, r_{(|E|+11)k+2}, \dots, r_{(|E|+11)k+n}\}$ . Thus  $G$  corresponds to the "net adjacency graph" [21] of the graph level.

To control the maximum number of asterisks that can be introduced in any column of  $M(G, q)$ , our reduction also has a *ballast* level  $B$ , an extension of the template and probe levels. First, define the function  $b: E \rightarrow \{0, 1, 2\}$  by

$$b(e) = \begin{cases} 2, & \text{if } e \in E - (E' \cup \{e_1\}) \\ 1, & \text{if } e \in (E' - \{e_0, e_2\}) \cup \{e_1\} \\ 0, & \text{if } e \in \{e_0, e_2\}. \end{cases}$$

For convenience, we ignore the row indexing. For each edge  $e \in E$ ,  $B$  contains a column  $B_e$  of  $kb(e)$  rows disjoint from the rows used in the columns above and disjoint from any other set  $B_{e'}$  corresponding to an edge  $e' \neq e$ . Thus, each row of  $B$  contains a single nonzero entry.

The matrix  $M(G, q)$  is obtained from the union of  $T$ ,  $P$ ,  $H$ , and  $B$  by a process of *identifying* certain columns. Two columns  $c_1$  and  $c_2$  described by row sets are identified by replacing  $c_1$  and  $c_2$  with the column represented by the union  $c_1 \cup c_2$ . The column identifications we require are as follows:

1. Identify  $t_0$  with  $p_5$ , and  $t_{|E'|+1}$  with  $p_1$ .
2. For  $i = 1, 2, \dots, |E'|$ , identify  $t_i$  with  $H_e$ , where  $e$  is the  $i$ th edge in the linear ordering  $\leq$  of  $E'$ .
3. Identify  $p_2$  with  $H_{e_2}$ , and  $p_4$  with  $H_{e_0}$ .
4. Identify  $p_3$  with  $H_{e_1}$ .
5. For each edge  $e \in E$ , identify  $H_e$  with  $B_e$ .

Since each component of  $G$  meets  $E' \cup \{e_1\}$ , the matrix  $M(G, q)$  is *connected*, in the sense that, for any two columns  $c$  and  $c'$ , there is a sequence of columns  $c = c_0, c_1, \dots, c_r = c'$  such that  $c_i \cap c_{i+1} \neq \emptyset$  for  $i = 0, 1, \dots, r-1$ . If  $c$  is a column of  $M(G, q)$  that is the result of identifying columns  $c_1$  and  $c_2$ , we may refer to  $c$  by either of the designations  $c_1$  or  $c_2$ . This is often convenient and should cause no confusion.

We now argue that  $M(G, q)$  has a GATE MATRIX LAYOUT cost of at most  $5k$  if, and only if,  $q \cdot G \in L$ . Suppose that the linear order  $\leq$  on  $E'$  extends to a linear order on  $E$  that witnesses  $q \cdot G \in L$ . Let  $\sigma_T = (t_0, t_1, \dots, t_{|E'|+1})$  be the sequence of columns of  $T$ , and let  $\sigma_P = (p_5, \dots, p_1)$  be the sequence of columns of  $P$ . The linear ordering  $\leq$  of  $E$  naturally describes a sequence  $\sigma_H$  of the columns of  $H$  with  $H_e$  occurring before  $H_{e'}$  in  $\sigma_H$  if, and only if,  $e \leq e'$ . The order  $\leq$  similarly describes a sequence  $\sigma_B$  of the columns of  $B$ . Let  $T' = T - \{t_0, t_{|E'|+1}\}$  and let  $\sigma_{T'} = (t_1, t_2, \dots, t_{|E'|})$ .

We make the following observation concerning our construction of  $M(G, q)$ :

- (\*) The sets of rows occurring in the columns of levels  $T$ ,  $P$ ,  $H$ , and  $B$  are pairwise disjoint.

Now consider a sequence  $\sigma'$  for  $M(G, q)$  that respects  $\sigma_{T'}$ ,  $\sigma_P$ ,  $\sigma_H$ , and  $\sigma_B$ . Thus,  $\sigma'$  describes a layout of  $T'$ ,  $P$ ,  $H$ , and  $B$  (no identifications). The subsequence  $\sigma_{T'}$  of  $\sigma'$  describes a "stairstep" layout of the columns of  $T'$  that contributes a cost of  $2k$  to each column of  $T'$  and a cost of  $k$  to any column occurring between two columns of  $T'$  in  $\sigma'$ . Similarly, the subsequence of  $\sigma_P$  of  $\sigma'$  describes a stairstep layout of the columns of  $P$  that contributes a cost of  $2k$  to each column of  $P$  and a cost of  $k$  to any column occurring between two columns of  $P$  in  $\sigma'$ . The subsequence  $\sigma_H$  of  $\sigma'$  contributes a cost of at most  $k$  to any column, and the subsequence  $\sigma_B$  similarly contributes a cost of at most  $2k$  to any column. By observation (\*), the cost incurred by any column, according to  $\sigma'$ , is the sum of the costs separately contributed by  $\sigma_{T'}$ ,  $\sigma_P$ ,  $\sigma_H$ , and  $\sigma_B$ .

Suppose  $\sigma$  is a sequence for  $M(G, q)$  that respects  $\sigma_T, \sigma_P, \sigma_H$ , and  $\sigma_B$  and with initial and final columns specified by  $(t_0, \dots, t_{|E|+1})$ . Our assumption that  $\leq$  extends to an order on  $E(G)$  that witnesses  $q.G \in L$  implies that we may choose  $\sigma$  so that columns that are identified in the construction of  $M(G, q)$  are consecutively adjacent in  $\sigma$ .

Consider now the effects of these identifications. Apart from  $t_0$  and  $t_{|E|+1}$  (each of which contains  $5k$  rows), each column of  $M(G, q)$  corresponds to a unique edge  $e$  of  $G$ . Let  $c_e$  denote this column. If  $e \in \{e_0, e_2\}$ , then  $\sigma_B$  contributes a cost of 0,  $\sigma_H$  contributes a cost of at most  $k$ , and each of  $\sigma_T$  and  $\sigma_P$  contributes a cost of  $2k$ . If  $e$  is in  $(E' - \{e_0, e_2\}) \cup \{e_1\}$ , then  $\sigma_B$  contributes a cost of  $k$ , one of  $\sigma_T$  and  $\sigma_P$  contributes a cost of  $k$  and the other contributes  $2k$ , and  $\sigma_H$  contributes a cost of at most  $k$ . If  $e$  does not belong to  $E' \cup \{e_1\}$ , then  $\sigma_B$  contributes a cost of  $2k$ , each of  $\sigma_T$  and  $\sigma_P$  contributes a cost of  $k$ , and  $\sigma_H$  contributes a cost of at most  $k$ . Thus, each column of  $M(G, q)$  under  $\sigma$  has a cost of at most  $5k$ .

Conversely, suppose there is a sequence  $\sigma$  of the columns of  $M(G, q)$  with a cost of at most  $5k$ . This implies that (up to symmetry) the first column of  $\sigma$  is  $t_0$  and the last is  $t_{|E|+1}$ , since  $M(G, q)$  is connected. It also implies that the restriction of  $\sigma$  to the columns of  $T$  must be equal to  $\sigma_T$  above, since (as in the proof of Theorem 2) any "kinking" would cause the width to be greater than  $5k$ . A similar remark applies to the restriction of  $\sigma$  to the columns of  $P$ . According to our construction of  $M(G, q)$  using column identifications, the column corresponding to  $e_0$  must occur before the column corresponding to  $e_1$  in  $\sigma$ , and this must occur before the column corresponding to  $e_2$ . Our argument is concluded by noting that the subsequences of  $\sigma$  (by restriction),  $\sigma_T, \sigma_P$ , and  $\sigma_B$  together contribute a cost of  $4k$  to every column except  $t_0$  and  $t_{|E|+1}$ . Thus, the restriction of  $\sigma$  to the columns of  $H$  represents a permutation with a cost of at most  $k$ . By the results of [8], this corresponds directly to an edge permutation demonstrating that  $q.G \in L$ .  $\square$

Since we have presented useful oracle languages and their associated scaffolding implementations in considerable detail for the two previous problems—one concerning vertex permutations and the other concerning edge permutations—we shall merely provide brief sketches of schemes that suffice for a few additional illustrative problems. For these and other problems amenable to this approach, we leave the low-level implementation details to the reader.

#### Theorem 4

For any fixed  $k$  (and  $d$ ), a satisfactory solution to MODIFIED MIN CUT, VERTEX SEPARATION, SEARCH NUMBER, and TWO-DIMENSIONAL GRID LOAD FACTOR can be constructed, if any exist, by an oracle algorithm [with overhead  $O(n^2)$ ] that makes  $O(n \log n)$  calls to an  $O(n^2)$  decision oracle.

*Proof sketch.* The last problem on this list is defined and the complexity of its decision version is addressed briefly in the Appendix. The other three problems are fairly cumbersome to define formally; the reader is referred to the specific sources listed in Section 3, where each is defined, or to [11], where each is defined and the quadratic bound on the complexity of each decision version is established.

An oracle algorithm for MODIFIED MIN CUT can be based on the oracle language we employed in the proof of Theorem 2. In addition, a scaffolding strategy for reducing the problem of recognizing this oracle language to the decision problem for MODIFIED

MIN CUT, with (fixed) parameter  $k' = 3k + 2$ , can be defined by modifying the scaffolding strategy we used there as follows:

1. Delete the vertices  $t_0$  and  $t_{|V|+1}$ .
2. Add four new vertices,  $a, b, c, d$ , and a set of  $3k + 2$  (multiple) edges joining each of the pairs of vertices  $(a, b)$ ,  $(a, t_1)$ ,  $(b, t_1)$ ,  $(t_{|V|}, c)$ ,  $(t_{|V|}, d)$ , and  $(c, d)$ .
3. Add additional edges so that for  $i = 1, 2, \dots, |V| - 1$  and for  $j = 1, \dots, 4$ , there is a set of  $k + 1$  edges joining  $t_i$  to  $t_{i+1}$  and  $p_j$  to  $p_{j+1}$ .

For VERTEX SEPARATION, we can again exploit the oracle language and the scaffolding strategy used in the proof of Theorem 2. In this case, we choose parameter  $k' = 3k$  and perform the following modifications:

1. Subdivide each edge once between  $t_i$  and  $t_{i+1}$  for  $i = 0, 1, \dots, |V|$ .
2. Subdivide each edge once between  $p_i$  and  $p_{i+1}$  for  $i = 1, 2, \dots, 4$ .

By Theorem 2.2 of [20], the search problem for SEARCH NUMBER reduces to the search problem for VERTEX SEPARATION. Thus, an oracle algorithm for SEARCH NUMBER may be described easily.

For the (fixed) parameter values  $k$  and  $d$ , an oracle algorithm for TWO-DIMENSIONAL GRID LOAD FACTOR can be based on an oracle language reflecting the "yes" instances of the following decision problem.

*Input:* A sextuple  $(G, f, d', n_0, n_1, v)$ , where  $G = (V, E)$  is a graph of order  $n$ ,  $f$  is a partial one-to-one map from  $V(G)$  to  $\{1, 2, \dots, d'\} \times \{1, 2, \dots, n\}$ ,  $1 \leq d' \leq d$ ,  $1 \leq n_0 \leq n_1 \leq n$ , and  $v \in V$ .

*Question:* Is there a one-to-one map  $F$  extending  $f$  to all of  $V$  that represents an embedding of  $G$  in the  $d \times n$  rectangular grid with a load factor of at most  $k$  and for which  $F(v) = (i, j)$  with  $i = d'$  and  $n_0 \leq j \leq n_1$ ?

A scaffolding construction that can be used to show that this problem is decidable in  $O(n^2)$  time is sketched as follows, using parameters  $k' = 3k$  and  $d' = d + 2$ .

The template level consists of a modified  $(d + 2) \times n$  rectangular grid. Each peripheral edge is replaced by a set of  $3k$  edges and every other edge is replaced by a set of  $2k$  edges, except for the edges in rows between the column indices  $n_0$  and  $n_1$ , which are replaced by a set of  $k$  edges each. The probe level consists of three vertices,  $u_0, u$ , and  $u_1$ , with  $k$  edges between  $u_0$  and  $u$ , and between  $u$  and  $u_1$ . The vertex  $u_0$  is identified with the grid vertex at coordinates  $(d', n_0)$ , and the vertex  $u_1$  is identified with the grid vertex at coordinates  $(d', n_1)$ . The vertex  $u$  is identified with the vertex  $v$ . The vertices of  $G$  are identified with the template vertices in a way that reflect the map  $f$ .  $\square$

## APPENDIX: NONCONSTRUCTIVE TOOLS AND THEIR APPLICATION

A *subdivision* of a graph  $H$  is any graph obtained from  $H$  by replacing edges with paths. Alternatively, one may view this as the insertion of some number of vertices of degree 2 into the edges of  $H$ . An example is illustrated in Figure A1.

A graph  $H$  is less than or equal to a graph  $G$  in the *topological* order, written  $H \leq_t G$ , if and only if a graph isomorphic to  $H$  can be obtained from  $G$  by a series of these two operations: taking a subgraph and *contracting* an edge at least one of whose endpoints has degree 2. A famous result of Kuratowski [26] states that a

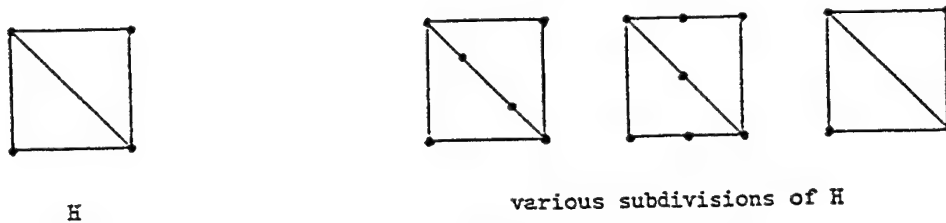
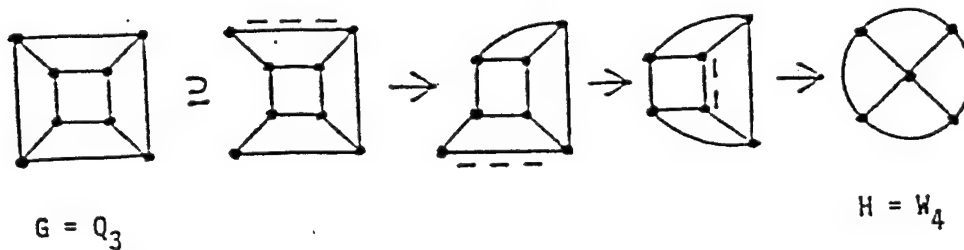


Figure A1. A graph and some of its subdivisions.



— — — contract

Figure A2. Construction demonstrating that  $W_4$  is a minor of  $Q_3$ .

graph  $G$  is nonplanar if, and only if,  $K_5 \leq_r G$  or  $K_{3,3} \leq_r G$ . The results we survey here can be viewed as a vast generalization of Kuratowski's Theorem.

A graph  $H$  is less than or equal to a graph  $G$  in the *minor* order, written  $H \leq_m G$ , if and only if a graph isomorphic to  $H$  can be obtained from  $G$  by a series of these two operations: taking a subgraph and contracting an arbitrary edge. For example, the construction depicted in Figure A2 shows that  $W_4 \leq_m Q_3$  (although  $W_4 \not\leq_r Q_3$ ).

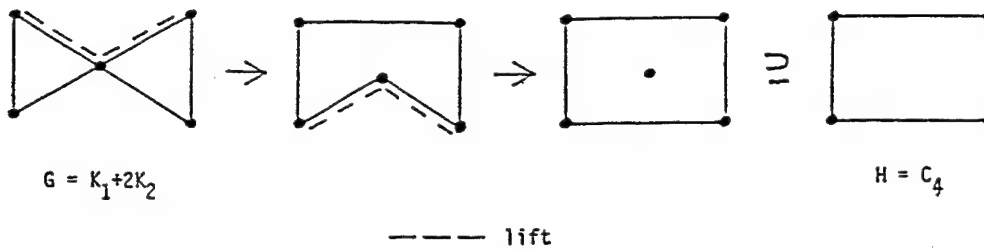
A family  $F$  of graphs is said to be *closed* under the minor ordering if the facts that  $G$  is in  $F$  and that  $H \leq_m G$  together imply that  $H$  must be in  $F$ . The *obstruction set* for a family  $F$  of graphs is the set of graphs in the complement of  $F$  that are minimal in the minor ordering. Therefore, if  $F$  is closed under the minor ordering, it has the following characterization:  $G$  is in  $F$  if, and only if, there exists no  $H$  in the obstruction set for  $F$  such that  $H \leq_m G$ .

#### Theorem A1 [7]

Any set of finite graphs contains only a finite number of minor-minimal elements.

#### Theorem A2 [6]

For every fixed graph  $H$ , the problem that takes as input a graph  $G$  and determines whether  $H \leq_m G$  is solvable in polynomial time.

Figure A3. Construction demonstrating that  $C_4$  is immersed in  $K_1 + 2K_2$ .

Theorems A1 and A2 guarantee only the *existence* of a polynomial-time decision algorithm for any minor-closed family of graphs. In particular, *no* proof of Theorem A1 can be entirely constructive [15]. Nevertheless, obstruction sets can often be isolated with problem-specific methods [16].

Letting  $n$  denote the number of vertices in  $G$ , the general time bound for algorithms ensured by these theorems is  $O(n^3)$ . If  $F$  excludes a planar graph, then the bound is  $O(n^2)$ . Much recent progress has been made to mitigate the enormous constants of proportionality first reported for such algorithms in [4]. New techniques greatly reduce the constants in general [17], and techniques specific to layout problems such as those we have addressed here lower them much more dramatically [15].

A graph  $H$  is less than or equal to a graph  $G$  in the *immersion* order, written  $H \leq_i G$ , if and only if a graph isomorphic to  $H$  can be obtained from  $G$  by a series of these two operations: taking a subgraph and *lifting* pairs of adjacent edges. For example, the construction depicted in Figure A3 shows that  $C_4 \leq_i K_1 + 2K_2$  (although  $C_4 \not\leq_m K_1 + 2K_2$  and  $C_4 \not\leq_r K_1 + 2K_2$ ).

The relation  $\leq_i$ , like  $\leq_m$ , defines a partial ordering on graphs with the associated notions of closure and obstruction sets.

#### Theorem A3 [3]

Any set of finite graphs contains only a finite number of immersion-minimal elements.

#### Theorem A4 [11]

For every fixed graph  $H$ , the problem that takes as input a graph  $G$  and determines whether  $H \leq_i G$  is solvable in polynomial time.

Theorems A3 and A4 also guarantee only the existence of a polynomial-time decision algorithm for any immersion-closed family of graphs. Our proof of Theorem A4 yields a general time bound of  $O(n^{h+3})$ , where  $h$  denotes the order of the largest graph in the relevant obstruction set. With knowledge of specific minors excluded by an immersion-closed family  $F$ , however, the time complexity for determining membership can, in many cases, be reduced to  $O(n^2)$  [11] by bounding the tree-width [5] of  $F$ .

For an application of Theorems A1 and A2, consider GATE MATRIX LAYOUT, a combinatorial problem arising in several VLSI layout styles, including gate matrix, programmable logic arrays under multiple folding, Weinberger arrays, and others. Although the general problem is  $\mathcal{NP}$ -complete, we have shown that, for any fixed value of  $k$ , an arbitrary instance can be mapped to an equivalent instance with only two 1s per column, then modeled as a graph on  $n$  vertices such that the family of "yes" instances is closed under the minor order and excludes a planar graph.

**Theorem A5 [8]**

For any fixed  $k$ , GATE MATRIX LAYOUT can be decided in  $O(n^2)$  time.

For an application of Theorems A3 and A4, consider TWO-DIMENSIONAL GRID LOAD FACTOR, a problem that is a two-dimensional analog of MIN CUT LINEAR ARRANGEMENT [12]. The *minimum load factor* of  $G$  relative to  $C$  is the minimum, over all embeddings of  $G$  in  $C$ , of the maximum number of paths in the embedding that share a common edge in  $C$ . In the TWO-DIMENSIONAL GRID LOAD FACTOR problem, we are given a graph  $G$  and integers  $k$  and  $w$ , and are asked whether the minimum load factor relative to an infinite-length, width- $w$  grid is less than or equal to  $k$ .

Although the general problem is  $\mathcal{NP}$ -complete even when  $w$  is fixed, we have shown that when both  $k$  and  $w$  are fixed, the family of "yes" instances is closed under the immersion order and has bounded tree-width.

**Theorem A6 [11]**

For any fixed  $k$  and  $w$ , TWO-DIMENSIONAL GRID LOAD FACTOR can be decided in  $O(n^2)$  time.

## REFERENCES

- [1] N. Robertson, and P.D. Seymour, "Disjoint Paths—A Survey," *SIAM Journal on Algebraic and Discrete Methods*, Vol. 6, pp. 300–305, 1985.
- [2] N. Robertson, and P.D. Seymour, "Graph Minors—A Survey," *Surveys in Combinatorics*, I. Anderson, ed., New York: Cambridge University Press, 1985, pp. 153–171.
- [3] N. Robertson, and P.D. Seymour, "Graph Minors IV. Tree-Width and Well-Quasi-Ordering," *Journal of Combinatorial Theory, Series B*, to appear.
- [4] N. Robertson, and P.D. Seymour, "Graph Minors V. Excluding a Planar Graph," *Journal of Combinatorial Theory, Series B* 41 pp. 92–114, 1986.
- [5] N. Robertson, and P.D. Seymour, "Graph Minors X. Obstructions to Tree-Decomposition," to appear.
- [6] N. Robertson, and P.D. Seymour, "Graph Minors XIII. The Disjoint Paths Problem," to appear.
- [7] N. Robertson, and P.D. Seymour, "Graph Minors XVI. Wagner's Conjecture," to appear.

- [8] M.R. Fellows, and M. A. Langston, "Nonconstructive Advances in Polynomial-Time Complexity," *Info. Proc. Letters*, Vol. 26, pp. 157-162, 1987.
- [9] M.R. Fellows, and M.A. Langston, "Nonconstructive Tools for Proving Polynomial-Time Decidability," *Journal of the ACM*, Vol. 35, pp. 727-739, 1988.
- [10] M.R. Fellows, and M.A. Langston, "Layout Permutation Problems and Well-Partially-Ordered Sets," *Proc. of 5th MIT Conference on Advanced Research in VLSI*, 1988, pp. 315-327.
- [11] M.R. Fellows, and M.A. Langston, "On Well-Partial-Order Theory and Its Application to Combinatorial Problems of VLSI Design," to appear.
- [12] M.R. Garey, and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, CA: Freeman, 1979.
- [13] D.J. Brown, M.R. Fellows, and M.A. Langston, "Polynomial-Time Self-Reducibility: Theoretical Motivations and Practical Results," *International Journal of Computer Mathematics*, Vol. 31, pp. 1-9, 1989.
- [14] F.S. Makedon, and I.H. Sudborough, "On Minimizing Width in Linear Layouts," *Lecture Notes in Computer Science*, Vol. 154, pp. 478-490, 1983.
- [15] M.R. Fellows, and M.A. Langston, "On Search, Decision and the Efficiency of Polynomial-Time Algorithms," *Proc. of 21st ACM Symposium on Theory of Computing*, 1989, pp. 501-512.
- [16] N.G. Kinnersley, "Obstruction Set Isolation for Layout Permutation Problems," PhD. thesis, Washington State University, Pullman, WA, 1989.
- [17] P.D. Seymour, private communication.
- [18] H.L. Bodlaender, "Improved Self-Reduction Algorithms for Graphs with Bounded Tree-Width," to appear.
- [19] W. Mader, "Hinreichende Bedingungen für die Existenz von Teilgraphen, die zu einem vollständigen Graphen homöomorph sind," *Math. Nachr.*, Vol. 53, pp. 145-150, 1972.
- [20] J. Ellis, I.H. Sudborough, and J. Turner, "Graph Separation and Search Number," to appear.
- [21] N. Deo, M.S. Krishnamoorthy, and M.A. Langston, "Exact and Approximate Solutions for the Gate Matrix Layout Problem," *IEEE Transactions on Computer-Aided Design*, Vol. 6, pp. 79-84, 1987.
- [22] T. Lengauer, "Black-White Pebbles and Graph Separation," *Acta Informatica*, Vol. 16, pp. 465-475, 1981.
- [23] T.D. Parsons, "Pursuit-Evasion in a Graph," *Theory and Application of Graphs*, Y. Alavi and D.R. Lick, eds. New York: Springer-Verlag, 1976, pp. 426-441.
- [24] M. Kirovski, and C.H. Papadimitriou, "Searching and Pebbling," *Theoretical Computer Science*, Vol. 47, pp. 205-218, 1986.
- [25] D.E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Reading, MA: Addison-Wesley, 1973.
- [26] C. Kuratowski, "Sur le Probleme des Courbes Gauches en Topologie," *Fund. Math.*, Vol. 15, pp. 271-283, 1930.



Michael R. Fellows was born in Upland, California, on June 15, 1952. He received his B.A. degree in Mathematics from Sonoma State University, Rohnert Park, California, in 1980, and his M.A. degree in Mathematics, in 1982, and his Ph.D. degree in Computer Science, in 1985, from the University of California, San Diego.

He has held positions at Washington State University, the University of New Mexico, and the University of Idaho. He is currently an Associate Professor in the Department of Computer Science at the University of Victoria, British Columbia, Canada. His research interests include combinatorial mathematics, the design and analysis of algorithms and computational complexity theory.

Dr. Fellows is a member of the American Mathematical Society, the Association for Computing Machinery, and the IEEE Computer Society.



Michael A. Langston was born on April 21, 1950, in Glen Rose, Texas. He received his B.S. degree in Mathematics from Texas A&M University in 1972, his M.S. degree in Systems and Information Science from Syracuse University in 1975, and his Ph.D. degree in Computer Science from Texas A&M University in 1981. From 1981 to 1989 he was on the faculty at Washington State University. Since then he has been on the faculty at the University of Tennessee.

Dr. Langston has authored or coauthored over fifty refereed papers, with publications appearing in *IEEE Transactions on Computers*, *Journal of the ACM*, *Operations Research*, *SIAM Journal on Computing*, and elsewhere. He has been an invited speaker at over forty universities, research laboratories, and international meetings. His present research interests include the design and analysis of algorithms, concrete complexity theory, graph theory, operations research, and VLSI.

He currently serves on the editorial boards of *Communications of the ACM*, *Parallel Processing Letters*, and *SIGACT News*. He is a member of the American Mathematical Society, the Association of Computing Machinery, the IEEE Computer Society, the Operations Research Society of America and the Society for Industrial and Applied Mathematics.

# Time-Space Optimal Parallel Merging and Sorting

Xiaojun Guan and Michael A. Langston

**Abstract**—A parallel algorithm is *time-space optimal* if it achieves optimal speedup and if it uses only a constant amount of extra space per processor even when the number of processors is fixed. Previously published parallel merging and sorting algorithms fail to meet at least one of these criteria. In this paper, we present a parallel merging algorithm that, on an EREW PRAM with  $k$  processors, merges two sorted lists of total length  $n$  in  $O(n/k + \log n)$  time and  $O(k)$  extra space, and is thus time-space optimal for any value of  $k \leq n/(\log n)$ . We also describe a *stable* version of our parallel merging algorithm that is similarly time-space optimal on an EREW PRAM. These two parallel merges naturally lead to time-space optimal parallel sorting algorithms.

**Index Terms**—Block rearranging, internal buffering, memory management, merging and sorting, parallel computation, time-space optimality.

## I. INTRODUCTION

THE quest for efficient parallel merging and sorting algorithms has been a long-standing topic of intense interest, as evidenced by the impressive volume of literature published on this subject (see, for example, [1], [6], [17] for recent surveys). Much of the focus has been on the search for methods that are *optimal* in the classic sense that asymptotically optimal speedup is attained.<sup>1</sup> Indeed, a number of parallel algorithms have been proposed that are optimal under this criterion, including those found in [3], [7]–[9], [15], [20], and [22].

Curiously, and quite unlike the case for sequential algorithms, very little attention seems to have been paid to space management issues. Some of this phenomenon can perhaps be attributed to the fact that much of what is known about parallel algorithms is relatively new. Accordingly, less time has elapsed for practical problems of implementation to become widely known. (See, for example, the formidable difficulties in memory management that have recently been encountered when an attempt has been made to implement

$\mathcal{NC}$ -style algorithms<sup>2</sup> on hypercubes with 16 and 256 nodes [5].) Another contributing factor may be that memory has become so inexpensive during the last few years that it is often easy simply to ignore it. In any event, space utilization continues to be a critical aspect in many applications, even for sequential processing; this criticality is only heightened in parallel processing systems when the number of processors is bounded.

None of the previously published parallel merging and sorting strategies are *time-space optimal*. That is, none achieve optimal speedup and, at the same time, require only a constant amount of extra space per processor even when the number of processors is fixed. We remark that, from a consideration of time alone, these algorithms represent an acceptable approach, mirroring one reason for the popularity of the parallel random-access machine (PRAM) model. Specifically, if the number of processors is fixed, then as the problem size grows, an  $\mathcal{NC}$  algorithm can be “scaled down,” so that each real processor needs merely to emulate multiple virtual processors, thereby accounting for the massive parallelism inherent in the design of the algorithm. Unfortunately, however, space requirements in this scenario tend to “blow up,” unless the extra space required by each real processor is constant, independent of the growing problem size. Added cause for concern is that, even if enough global memory is available, the more shared memory accesses a program makes the more message traffic is placed on whatever interconnection network is used to realize the shared memory, with an attendant downgrading of the overall system’s performance. Incorporating secondary memory devices into this picture naturally leads to additional problems [18], to be avoided as long as main memory need not be squandered on temporary extra storage.

From the foregoing discussion, we conclude that any genuine attempt to minimize extra space dictates that the total number of extra storage cells required by each processor be constant, even when the number of processors available is bounded by some constant  $k$  whose value is independent from the size of a problem instance,  $n$ . (This is in contrast to work such as that described in [19], in which constant extra space is employed at each processor, but the number of processors is assumed to be  $\Theta(n^2)$ .) Moreover, as an attractive side effect of attempting to minimize extra space, a bounded number of processors reflects more faithfully any real parallel computing environment.

In this paper, we present for the first time a parallel merging algorithm that, on an exclusive-read exclusive-write (EREW)

Manuscript received January 1, 1989; revised April 27, 1990. A preliminary version of this paper was presented at the International Conference on Parallel Processing, St. Charles, IL, Aug. 1989.

X. Guan is with the Department of Computer Science, Washington State University, Pullman, WA 99164. This author’s work was supported in part by the Washington State University Graduate Research Assistantship Program.

M. A. Langston is with the Department of Computer Science, University of Tennessee, Knoxville, TN 37996 and the Department of Computer Science, Washington State University, Pullman, WA 99164. This author’s work was supported in part by the National Science Foundation under Grants MIP-8603879 and MIP-8919312, and by the Office of Naval Research under Contract N00014-88-K-0343.

IEEE Log Number 9143276.

<sup>1</sup>A parallel method attains asymptotically optimal speedup if the product of the number of processors it employs and the amount of time it takes is within a constant factor of the time required by a fastest sequential algorithm.

<sup>2</sup>A problem is said to be in  $\mathcal{NC}$  if it possesses a parallel algorithm that, for any problem instance of size  $n$ , employs a number of processors bounded by some polynomial function of  $n$  and requires an amount of time bounded by some polylogarithmic function of  $n$ .

PRAM, merges two sorted lists in  $O(n/k + \log n)$  time and constant extra space per processor, and hence is time-space optimal for any value of  $k \leq n/(\log n)$ . We also describe how this gives rise to a stable<sup>3</sup> version of our parallel merging algorithm that is similarly time-space optimal on an EREW PRAM. We observe that (our technique for achieving) stability incurs two penalties: a slightly more complicated algorithm and somewhat larger constants of proportionality. These two parallel merges naturally lead to time-space optimal parallel sorting algorithms.

In the next section, we briefly review the main features of a recently-published linear-time in-place sequential merge. Although a direct parallelization of this method is not possible, its overall structure is helpful in simplifying the presentation of our parallel algorithm, which we describe in detail in Section III. As we demonstrate in that section, a major factor in our algorithm's asymptotic time-space optimality is the introduction of a useful technique that is based on what we dub a *displacement table*. We next move on to the subject of stable merging, describing in Section IV how some relatively simple modifications to our parallel merge can be exploited to yield a stable time-space optimal parallel algorithm. Extensions to sorting and open topics for future research are discussed in the final section.

## II. A REVIEW OF TIME-SPACE OPTIMAL SEQUENTIAL MERGING

To simplify the presentation of our time-space optimal parallel algorithm in the next section, it is useful first to review at least briefly the recently-published and relatively simple linear-time, in-place sequential merge from [10]. Expectedly, some of the operations that are easy to perform sequentially are difficult to perform in parallel. Interestingly, on the other hand, some of the operations that are difficult to perform sequentially are easy to perform in parallel. On the whole, however, it turns out that a direct parallelization of this novel sequential method is not possible. Nevertheless, its overall structure can be used to *guide our thinking* so that, with the aid of our parallel displacement table technique to be presented later, we can direct all available processors to work efficiently in unison.

The (sequential) optimality attained with respect to both time and space inherently relies on the related notions of *block rearranging* and *internal buffering*. To get a feel for the general way in which such a strategy works, it is helpful to view a list containing  $n$  records as a collection of  $\Theta(\sqrt{n})$  blocks, each of size  $\Theta(\sqrt{n})$ . This approach allows us to employ one block as the (internal) buffer to aid in resequencing the other blocks of the two sorted sublists and then merging these blocks into one sorted list. Since only the contents of the buffer and the relative order of the blocks need ever be out of sequence, linear time is sufficient to achieve order by straight-selection sorting [14] both the buffer and the blocks (each sort involves  $O(\sqrt{n})$  keys). We refer the interested reader to [10]–[13] for

extensive background, related results, and additional details on block rearranging and internal buffering methods.

We note that, for the sake of complete generality, we allow neither the key nor any other part of a record to be modified by our algorithms. Such is necessary, for example, when records are write-protected or when there is no explicit key field within each record, but instead a record's key is a function of one or more of its data fields.

Let  $L$  denote a list containing two sublists to be merged, each with its keys in nondecreasing order. We shall make a few simplifying assumptions about  $L$  to facilitate the discussion. (See [10] for a complete exposition of the algorithm, an example, and the  $O(\sqrt{n})$  time and  $O(1)$  space implementation details necessary for handling arbitrary inputs.)

We assume that  $n$  is a perfect square, and that the records of  $L$  have already been permuted so that  $\sqrt{n}$  largest-keyed records are at the front of the list (their relative order there is immaterial), followed by the remainders of the two sublists, each of which we now assume contains an integral multiple of  $\sqrt{n}$  records in nondecreasing order. Therefore, we can view  $L$  as a series of  $\sqrt{n}$  blocks, each of size  $\sqrt{n}$ . The leading block will be used as an internal buffer to aid in the merge.

The first step is to sort the  $\sqrt{n} - 1$  rightmost blocks by their *tails* (rightmost elements), after which their tails form a nondecreasing key sequence. (In this setting, selection sort requires only  $O(n)$  key comparisons and record exchanges.) Records within a block retain their original relative order.

The second step, which is the most complex, is to direct a sequence of series merges. An initial pair of series of records to be merged is located as follows. The first series begins with the head of block 2 and terminates with the tail of block  $i$ ,  $i \geq 2$ , where block  $i$  is the first block such that the key of the tail of block  $i$  exceeds the key of the head of block  $i + 1$ . The second series consists solely of the records of block  $i + 1$ . The buffer is used to merge these two series. That is, the leftmost unmerged record in the first series is repeatedly compared to the leftmost unmerged record in the second, with the smaller-keyed record swapped with the leftmost buffer element. Ties are broken in favor of the leftmost series. (In general, the buffer may be broken into two pieces as the merge progresses.) This task is halted when the tail of block  $i$  has been moved to its final position.

The next two series of records to be merged are now located. This time, the first begins with the leftmost unmerged record of block  $i + 1$  and terminates as before for some  $j \geq i$ . The second consists solely of the records of block  $j + 1$ . The merge is resumed until the tail of block  $j$  has been moved. This process of locating series of records and merging them is continued until a point is reached at which only one such series exists, which is merely shifted left, leaving the buffer in the last block.

The final step is to sort the buffer, thereby completing the merge of  $L$ .

$O(n)$  time suffices for this entire procedure, because each step requires at most linear time.  $O(1)$  space suffices as well, since the buffer was internal to the list, and since only a handful of additional pointers and counters are necessary.

<sup>3</sup>A merging algorithm is stable if it preserves the original relative order of records with identical keys.

### III. TIME-SPACE OPTIMAL PARALLEL MERGING ON THE EREW PRAM MODEL

Note that, exclusive of implementation details for extracting the internal buffer and for handling lists and sublists of arbitrary sizes, the sequential algorithm just described comprises three steps: *block sorting*, *series merging*, and *buffer sorting*. Unfortunately, these steps do not appear to permit a direct parallelization, at least not one that requires only constant extra space per processor. In particular, the internal buffer is instrumental in the series merging step, dictating a block size of  $\Theta(\sqrt{n})$  that in turn severely limits what can be accomplished efficiently in parallel.

Optimistically, however, observe that if we could only devise a time-space optimal method to

- 1) use bigger blocks (namely, one block for each of the  $k$  processors, giving rise to a block size of  $n/k$ , a value that might be unboundedly greater than  $\sqrt{n}$ ) and
- 2) reorganize the file so that the problem is reduced to one of  $k$  local merges (that is, replace the contents of each block with two sublists, one from each of the two original sublists in  $L$ , so that the largest key in block  $i$  is no greater than the smallest key in block  $i + 1$ , for  $1 \leq i < k$ ),

then we could complete a time-space optimal merge of  $L$  by simply directing each processor to merge the contents of its own block using the algorithm sketched in the last section. This observation is the genesis of the parallel method we shall now present.

Ignoring for the moment implementation details for dealing with lists and sublists of arbitrary sizes (these details will be addressed at the end of this section), our parallel method comprises these five steps: *block sorting*, *series delimiting*, *displacement computing*, *series splitting*, and *local merging*. Since the last step of our algorithm (local merging) is easy from a parallel standpoint, it is not surprising that the earlier steps are relatively complicated, requiring a careful coordination of all processors to achieve efficiently the desired reorganization of the file.

To facilitate discussion, let us temporarily assume that the number of records in each of the two sublists in  $L$  is evenly divisible by  $k$ . We shall refer to a record or block from the first sublist of  $L$  as an  $L1$  record or an  $L1$  block. We shall use the terms  $L2$  record and  $L2$  block in an analogous fashion for elements from the second sublist.

**Block Sorting:** We first view  $L$  as a sequence of  $k$  blocks, each of size  $n/k$ . See Fig. 1, in which we employ a handy pictorial representation for  $L$ , using the vertical axis to indicate increasing key values and the horizontal axis to indicate increasing record indexes. We seek to sort these blocks by their tails. This is a relatively simple chore if one is willing to settle for a concurrent-read exclusive-write (CREW) algorithm. For example, we could begin by directing each  $L1$  ( $L2$ ) processor to perform a binary search on the  $L2$  ( $L1$ ) sublist, comparing its block's tail against the tails in that sublist. In order to sort the blocks efficiently on the EREW model, we adopt a slightly more complex strategy.



Fig. 1. Divide lists into blocks.

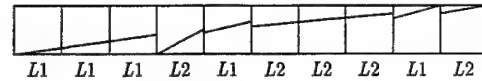


Fig. 2. Block sorting.

We first direct each processor to set aside a copy of the tail of its block and its index (an integer between 1 and  $k$ , inclusive). We can now merge the  $k$  tail copies (dragging along the indexes) by reversing in parallel the copies from the second sublist and then invoking the well-known bitonic merge [4], a task requiring  $O(\log k)$  time and  $O(k)$  total extra space.

After this merge is completed, each processor knows the index of the block it is to receive. With the use of but one extra storage cell per processor, it is now a simple matter for the processors to acquire their respective new blocks in parallel without memory conflicts, one record at a time (say, from the first record in a block to the last). This task requires  $O(n/k)$  time and  $O(k)$  extra space.

This completes the block sorting step, and has required  $O(n/k + \log k)$  time and constant extra space per processor. See Fig. 2.

**Series Delimiting:** As with the sequential method, it is helpful at this point to think of the list as containing a collection of pairs of series of records, with each pair of series to be merged. (Of course, we cannot now merely mimic the sequential series merging step. If there are large series, then it would take too long to merge them; if there are large blocks, then it would take too long to sort any type of internal buffer.) We require a somewhat more refined definition of "series," however, because we must insist that pairs of series do not overlap one another. The first and second series of any given pair meet as before, where the tail of block  $i$  exceeds the head of block  $i + 1$ . To determine where pairs meet each other, we now use the term "breaker" to denote the first record of block  $i + 1$  that is no smaller than the tail of block  $i$ . Thus, the first series of a pair needs only to begin with a breaker, and the second series of that pair needs only to end with the record immediately preceding the next breaker. This notion is illustrated in Fig. 3. Because each pair of series is made up either of a portion of an  $L1$  block followed by zero or more full  $L1$  blocks and a portion of an  $L2$  block, or a portion of an  $L2$  block followed by zero or more full  $L2$  blocks and a portion of an  $L1$  block, and because these two configurations are symmetric, we shall henceforth address only the former case in this and subsequent figures.

For a processor to determine whether its block contains a second series, it simply compares its head to its left neighbor's tail. If this comparison reveals that the processor does contain such a series, then it invokes a binary search to locate its breaker (it must have one—recall that the blocks were first

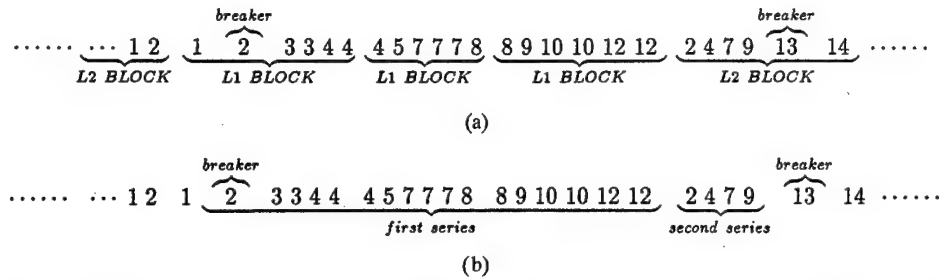


Fig. 3. Delimiting a pair of series to be merged. (a) Locating the breakers. (b) Pair of resulting series.

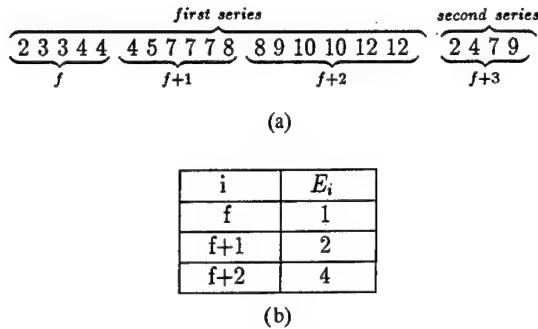


Fig. 4. A pair of series and the corresponding displacement table entries. (a) Pair of series,  $p = 3$ . (b) Displacement table entries.

sorted by their tails) and broadcasts<sup>4</sup> the breaker's location first to its left and then to its right. By this means, a processor learns the location of the breaker to its immediate right and the location of the breaker to its immediate left.

From this it follows that every processor can correctly delimit the one or two pairs of series that are relevant to the contents of its block in  $O(\log(n/k) + \log k)$  time and constant extra space per processor.

**Displacement Computing:** Recall that our goal is to reorganize the file so that local merging is possible as a final step. This requires an efficient parallel means for splitting each pair of series among the processors that are in charge of the pair's blocks. In order to accomplish this, we shall now introduce what we term a *displacement table*, with one table entry to be stored at each processor. In this table we seek to enumerate, for each processor with a block (or portion thereof) from the first series, the number of records from the second series that would displace records in that block if there were no other records in the first series. Thus, a displacement table is of immediate use in the next step (series splitting), because processor  $i$  needs only to know its entry,  $E_i$ , and the entry for processor  $i - 1$ ,  $E_{i-1}$ . From these two values it is easy for processor  $i$  to determine the number of its records that are to be displaced by records from the left (namely,  $E_{i-1}$ ) and the number that are to be displaced by records from the second series (namely,  $E_i - E_{i-1}$ ). See Fig. 4.

As with the block sorting step, things are relatively simple if one is willing to settle for a CREW algorithm. For example, we could begin by directing each processor whose block

contains records from the first series to perform a binary search on the second series. In order to compute the displacement table entries efficiently on the EREW model, we adopt a considerably more complicated strategy. In particular, we must solve a nontrivial *processor allocation problem* [20]. We agree with the sentiment expressed by others (see, for example, [7], [16]) that details relevant to this thorny subject warrant a careful exposition.

For an arbitrary pair of series, let  $f$  denote the index of the processor handling the first record in the first series, and let  $p$  denote the number of blocks with records in that series. Thus, processor  $f + p$  is responsible for the second series. We seek to direct the  $p$  processors with records in the first series to work in unison and without memory conflicts to determine where each of their block's tails would need to go if they were merged with the  $m < n/k$  records of the second series. To accomplish this, we now present a technique that is perhaps best described as a sequence of phases of operations.

In the first phase, each processor with records in the first series sets aside a copy of its block's tail and its index (an integer between  $f$  and  $f + p - 1$ , inclusive). Each also sets aside two pieces of information from the second series: processor  $i$  ( $f \leq i < f + p$ ) computes and saves a copy of the offset  $h = (i - f + 1)(m/p)$  and a copy of the  $h$ th record of the second series. We can now merge the  $2p$  elements made up of  $p$  tails and  $p$  selected records (dragging along the indexes and the offsets) by reversing in parallel the selected records and then invoking a bitonic merge, a task requiring  $O(\log p)$  time and  $O(p)$  extra space.

After this, each processor with records in the first series examines the two keys in its temporary storage. If a processor finds a tail, then (with the use of the tail's index) it reports its own index to the processor handling the block from which the tail originated. Thus, every processor can determine from the movement of its block's tail just how many of the records selected from the second series are smaller, and therefore which of the  $p$  subseries of the second series, each subseries of size  $m/p$ , to merge into next. In order for a processor to be able to determine how many other tails are to be merged into the same next subseries as its block's tail, each one compares its next subseries to that of its neighbors. If the comparison reveals a subseries boundary, then broadcasting is used to inform the other processors of the location of this boundary (as we did when broadcasting a breaker's location in the series delimiting step).

For the second and each subsequent phase, processors

<sup>4</sup>A convenient algorithm for this type of broadcasting can for example be found in [21, p. 234], where it is termed a "data distribution algorithm." Alternately, such broadcasting can be efficiently accomplished with parallel prefix computation.

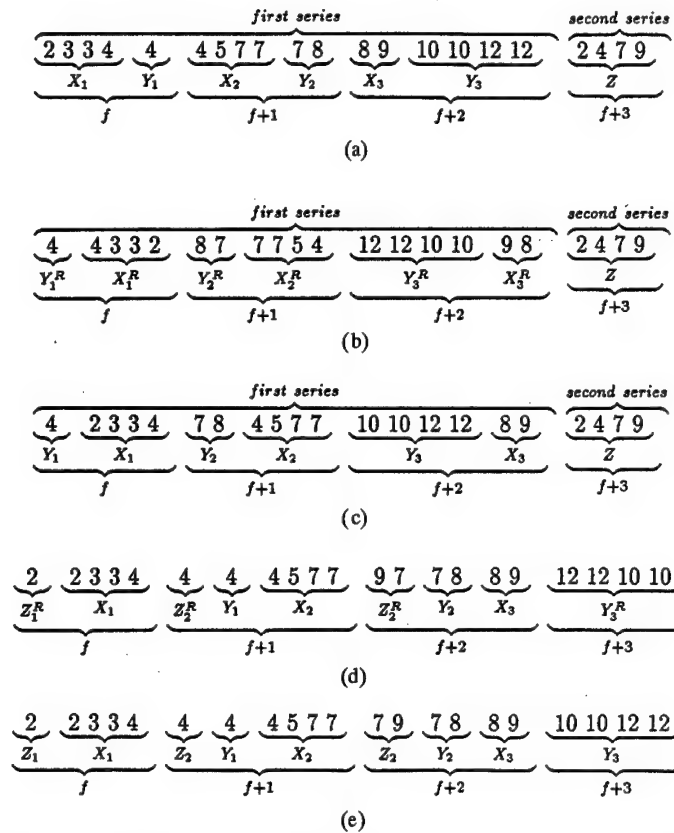


Fig. 5. Series splitting. (a) Notation. (b) Block rotation. (c) Subblock rotation. (d) Data movement. (e) Subblock rotation.

proceed as in the first phase, but now with new offsets and selected records based on the proper subseries into which their block's tails are to be merged and the number of other tails that are also to be merged there. Processors continue to iterate this procedure until each has determined where its block's tail would go if it were merged with the other tails and the second series. Note that some processors may be employed in as few as  $\log_k m$  phases, each requiring  $O(\log k)$  time, while others may simultaneously be employed in as many as  $\log_2 m$  phases, each requiring constant time. In general, letting the sequence  $k_1, k_2, \dots, k_l$  denote the number of tails in any chain of recursive calls, we observe that  $k_1 \times k_2 \times \dots \times k_l$  is  $O(m)$ , and hence  $\log k_1 + \log k_2 + \dots + \log k_l$  is  $O(\log m)$ . Therefore,  $O(\log n)$  time and  $O(k)$  extra space have been consumed up to this point.

Let  $l_i (1 \leq l_i \leq m + p)$  denote the location that the tail of the block of processor  $i$  ( $f \leq i < f + p$ ) would occupy in a sublist containing the  $p$  tails and the entire second series if such a sublist were available. Processor  $i$  now computes  $l'_i = l_i - (i - f) - 1$ , to eliminate the effect of its block's tail and all preceding tails. It next employs two pointers to compare a record in its block, beginning at location  $n/k$  (its tail), to a record in the second series, beginning at location  $l'_i$ , repeatedly decrementing the pointer that points to the larger key for  $l'_i$  iterations. (We insist that each processor works from right to left in its interval of the second series in order to avoid memory conflicts, and that processor  $i$  keeps track of  $l'_{i-1}$  and  $l'_{i+1}$ , relying on broadcasting by the leftmost processor if degeneracy in an interval occurs.) When processor  $i$  has

finished decrementing its two pointers in this fashion, a task requiring  $O(n/k)$  time and  $O(k)$  extra space, the value of its second series pointer is its displacement table entry,  $E_i$ .

Thus, displacement computing can be accomplished in  $O(n/k + \log n)$  time and constant extra space per processor.

**Series Splitting:** At this point, processor  $i$  can easily determine from the entries in the displacement table the number of its records that are to be displaced to the block to its right ( $E_i$ ), as well as the number of records that it is to receive from the block to its left ( $E_{i-1}$ ) and from the second series ( $E_i - E_{i-1}$ ). Thus, we now seek to split, in parallel, the second series among the blocks of the first series. We accomplish this efficiently in constant extra space with the use of block rotations (each of which is effected with a sequence of three sublist reversals), followed by the desired data movement, followed by one last reversal. We illustrate this procedure in Fig. 5, with the aid of some additional notation.

Letting  $i$  denote the index of an arbitrary processor with records in the first series only, we use  $X_i$  to denote its first  $n/k - E_i$  records (that is, those to remain in this block) and  $Y_i$  to denote the remaining  $E_i$  records (that is, those to be displaced to the right). We use  $Z$  to denote the contents of the portion of a block that constitutes the second series. Processor  $i$  first reverses  $X_i$  and  $Y_i$  together, then each separately, thereby completing the rotation. Processor  $i$  then initiates data movement, employing a single extra storage cell to copy safely the last record of  $Y_i$  to the location formerly occupied by the last record of  $Y_{i+1}$ . We deviate from this if processor  $i$  is handling the last block of the first series, instructing it

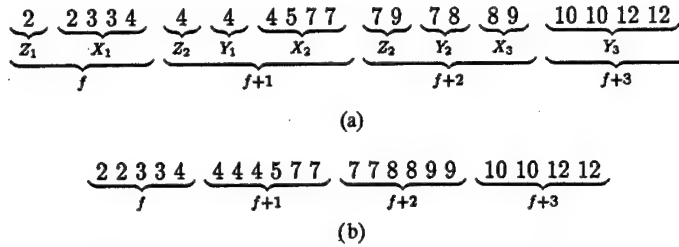


Fig. 6. Local merging. (a) Series splitting completed. (b) Local merging.

instead to copy its last  $Y$  record to the former location of the first  $Z$  record. At the same time, the processor of the second series copies its first  $Z$  record to the former location of the last  $Y$  record of the first (portion of a) block in the first series. Continuing in this fashion, therefore, the data movement sequence is right-to-left for the blocks in the first series, but left-to-right for the second.

Of course, when block  $i$  of the first series is filled, the processor of the second block must shift its attention to block  $i + 1$ , and so on. If  $k$  is small enough [no greater than  $O(\log n)$ ], then the displacement table can simply be searched; if  $k$  is larger than this, then the table may contain too many identical entries, and we invoke a preprocessing routine to condense it (again with the aid of broadcasting). The timing of the first and second series operations are interleaved (rather than simultaneous), because some processors will in general be handling portions of blocks of both types of series.

When the data movement phase is finished, each block will contain the correct prefix from the opposite series, but in reverse order. A final subblock reversal completes this step.

Series splitting, therefore, requires  $O(n/k + \log n)$  time and constant extra space per processor.

**Local Merging:** We employ the aforementioned linear-time, in-place sequential merge from [10]. The completion of this merge is depicted in Fig. 6.

Thus, this final step requires  $O(n/k)$  time and constant extra space per processor.

**Implementation Details:** Although the details necessary to handle lists and sublists of arbitrary sizes is the most intricate part of the sequential method, these details are quite simple for our parallel algorithm. We first fragment the input list  $L = L1L2$  into the form  $L3L4L5L6$ , where both  $L3$  and  $L5$  contain an integral multiple of  $n/k$  records, and where  $L4$  and  $L6$  each contain strictly less than  $n/k$  (even, possibly, zero) records. With parallel rotations, it is easy to transform the list into the form  $L3L5L4L6$  assuming the tail of  $L4$  is less than or equal to the tail of  $L6$  (or the form  $L3L5L6L4$  if it is greater). We now invoke the main parallel algorithm on  $L3L5$ , yielding the sorted sublist  $L7$ . Ignoring obvious ways to streamline the remainder of this procedure, it is sufficient at this point merely next to invoke the sequential algorithm on  $L4L6$ , yielding the sorted sublist  $L8$ . Thus,  $L8$  can be viewed as at most one block of size  $n/k$  followed by at most one block of size strictly less than  $n/k$ . We now complete the merge by invoking the main parallel algorithm on  $L7L8$ , with every processor except possibly the last handling a block of size  $n/k$ . Even though the last block may have an unusual

size at this step, it causes no problems for the main algorithm because its (large) tail ensures that it need not be moved during block sorting and because its (rightmost) position ensures that it need not be treated as a member of a first series when any pair of series is merged.

The time and space requirements necessary for implementation details are therefore bounded by those of the main parallel algorithm.

This completes the description of our parallel method. In summary, the total time spent is  $O(n/k + \log n)$  and the total extra space used is  $O(k)$ . Therefore, this method is time-space optimal for any value of  $k \leq n/(\log n)$ , thereby meeting our stated goal.

#### IV. ENSURING STABILITY

It is often desirable that merging (and sorting) algorithms be *stable*, by which we mean that records with identical keys retain their original relative order after the algorithm is completed. Stability is a property that has exacted a heavy price in terms of increased complexity for sequential algorithms that operate in both optimal time and space simultaneously. The linear-time, in-place stable sequential merging algorithm with the lowest currently-known worst case constant of proportionality is presented in [11] and is based largely on the unstable method that proved useful in guiding our thinking in devising the parallel algorithm presented in the last section. As one rough measure of the intricacy required to ensure stability in a sequential setting, we note that the worst case constant of proportionality jumps from  $3.125n$  (plus lower order terms) for the unstable algorithm of [10] to  $7n$  (plus lower order terms) for the stable scheme of [11], where these values reflect an upper bound on the number of key comparisons plus record exchanges required.

Fortunately, however, the parallel procedure we have already presented can be made stable with relatively little effort. The only unstable routines in our main algorithm are found in the steps for block sorting, displacement computing, and local merging. Instability in the block sorting step can be remedied by stabilizing the bitonic merge. To accomplish this, we need only specify that the block indexes (which are already available) are to be used as "tie breakers" whenever equal tails are compared. The displacement computing step can be modified in a similar manner, by first stabilizing the bitonic merge (indexes and offsets are already available) and then handling the two (now asymmetric) types of pairs of series in slightly different fashions in that  $L1$  records must now receive priority over  $L2$  records. The local merging step is stabilized by

replacing the relatively simple but unstable in-place algorithm with the more complicated but stable in-place scheme. Only an extra pointer is needed to stabilize the implementation details (in the event that the  $L3$  and  $L4$  sublists each have a copy of the same key).

## V. EXTENSIONS TO SORTING AND OPEN PROBLEMS

In this paper, we have presented for the first time parallel merging algorithms that are asymptotically time-space optimal. Moreover, our methods assume only the EREW PRAM model. Although  $n$  must be large enough so that the inequality  $k \leq n/(\log n)$  is satisfied for optimality, we observe that our algorithms are efficient<sup>5</sup> for any value of  $n$ , suggesting that they may have practical merit even for relatively small inputs. Also, for the sake of complete generality, our algorithms modify neither the key nor any other part of a record.

These time-space optimal parallel merging algorithms naturally lead to time-space optimal parallel sorting algorithms, providing improvements over the best previously-published PRAM methods designed for a bounded number of processors. For example, the recent EREW merging and sorting schemes proposed in [3] (where the issue of duplicate keys is not even addressed) are time optimal only for values of  $k \leq n/(\log^2 n)$ . More importantly, such schemes are not space optimal for any fixed  $k$ .

We note that, from a practical standpoint, more streamlined sorting implementations may be possible. It is known from [11] that methods exist by which the obvious merge sort strategy can be replaced with more sophisticated sorting schemes that exploit merging in nontrivial ways. In that setting, for example, the worst case constant of proportionality of the direct merge sort strategy is lowered from  $7n \log n$  (plus lower order terms) to  $2.5n \log n$  (plus lower order terms). Whether these more complicated techniques can be efficiently parallelized remains an open question.

Finally, from a more purely theoretical perspective, one might ask whether our methods can be extended to sublogarithmic time merging. Because  $\Omega(\log n)$  time is known to be a lower bound for merging on an EREW PRAM, our algorithms are the best possible (to within a constant factor) for this model. Asymptotically faster time-space optimal algorithms may exist, however, for more powerful models. For example, it is an open question whether time-space optimal merging can be accomplished in  $O(n/k + \log \log n)$  time on a CREW PRAM.

## ACKNOWLEDGMENT

We wish to express our appreciation to K. Abrahamson and

B.-C. Huang for stimulating technical discussions related to this general subject, and to the two anonymous referees for comments that have helped to improve the presentation of these results.

## REFERENCES

- [1] S. G. Akl, *Parallel Sorting Algorithms*. Orlando, FL: Academic, 1985.
- [2] M. Ajtai, J. Komlos, and E. Szemerédi, "An  $O(n \log n)$  sorting network," in *Proc. 15th ACM Symp. Theory Comput.*, 1983, pp. 1-9.
- [3] S. G. Akl and N. Santoro, "Optimal parallel merging and sorting without memory conflicts," *IEEE Trans. Comput.*, vol. C-36, pp. 1367-1369, 1987.
- [4] K. E. Batchier, "Sorting networks and their application," in *Proc. AFIPS 1968 SJCC*, 1968, pp. 307-314.
- [5] P. Banerjee and K. P. Belkhale, "Parallel algorithms for geometric connected component labeling problems on a hypercube," Tech. Rep., Coordinated Sci. Lab., Univ. of Illinois, Urbana, IL, 1988.
- [6] D. Bitton, D. J. Dewitt, D. K. Hsiao, and J. Menon, "A taxonomy of parallel sorting," *Comput. Surveys*, vol. 16, pp. 287-318, 1984.
- [7] A. Borodin and J. E. Hopcroft, "Routing, merging and sorting on parallel models of computation," *J. Comput. Syst. Sci.*, vol. 30, pp. 130-145, 1985.
- [8] G. Baudet and D. Stevenson, "Optimal sorting algorithms for parallel computers," *IEEE Trans. Comput.*, vol. C-27, pp. 84-87, 1978.
- [9] R. Cole, "Parallel merge sort," *SIAM J. Comput.*, vol. 17, pp. 770-785, 1988.
- [10] B.-C. Huang and M. A. Langston, "Practical in-place merging," *Commun. ACM*, vol. 31, pp. 348-352, 1988.
- [11] —, "Fast stable merging and sorting in constant extra space," in *Proc. 1989 Int. Conf. Comput. Inform.*, 1989, pp. 71-80.
- [12] —, "Stable duplicate-key extraction with optimal time and space bounds," *Acta Informatica*, vol. 26, pp. 473-484, 1989.
- [13] —, "Stable set and multiset operations in optimal time and space," in *Proc. 7th ACM Symp. Principles Database Syst.*, 1988, pp. 288-293.
- [14] D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Reading, MA: Addison-Wesley, 1973.
- [15] C. P. Kruskal, "Searching, merging and sorting in parallel computation," *IEEE Trans. Comput.*, vol. C-32, pp. 942-946, 1983.
- [16] R. M. Karp and V. Ramachandran, "A survey of parallel algorithms for shared-memory machines," Tech. Rep., Comput. Sci. Division, Univ. of California, Berkeley, CA, 1988.
- [17] S. Lakshmivarahan, S. K. Dhall, and L. L. Miller, "Parallel sorting algorithms," *Advances Comput.*, vol. 23, pp. 295-354, 1984.
- [18] D. Rose and F. Berman, "Mapping with external I/O: A case study," in *Proc. 1987 Int. Conf. Parallel Processing*, 1987, pp. 859-862.
- [19] S. Ranka and S. Sahni, "Image template matching on SIMD hypercube multicomputers," in *Proc. 1988 Int. Conf. Parallel Processing*, 1988, vol. 3, pp. 84-91.
- [20] Y. Shiloach and U. Vishkin, "Finding the maximum, merging and sorting in a parallel computation model," *J. Algorithms*, vol. 2, pp. 88-102, 1981.
- [21] J. D. Ullman, *Computational Aspects of VLSI*. Rockville, MD: Computer Science Press, 1984.
- [22] L. G. Valiant, "Parallelism in comparison problems," *SIAM J. Comput.*, vol. 4, pp. 349-355, 1975.

Xiaojun Guan, photograph and biography not available at the time of publication.

Michael A. Langston, photograph and biography not available at the time of publication.

<sup>5</sup>A parallel method is efficient if its speedup is within a polylogarithmic factor of the optimum.

## POLYNOMIAL-TIME SELF-REDUCIBILITY: THEORETICAL MOTIVATIONS AND PRACTICAL RESULTS\*

DONNA J. BROWN

*Coordinated Science Laboratory, University of Illinois, Urbana, IL 61801, USA*

MICHAEL R. FELLOWS

*Department of Computer Science, University of Idaho, Moscow, ID 83843, USA*

MICHAEL A. LANGSTON

*Department of Computer Science, Washington State University, Pullman,  
WA 99164-1210, USA*

(Received 30 September 1988)

Although polynomial-time complexity theory has been formulated in terms of decision problems, polynomial-time decision algorithms generally operate by attempting to construct a solution to an optimization version of the problem at hand. Thus it is that *self-reducibility*, the process by which a decision algorithm may be used to devise a constructive algorithm, has until now been widely considered a topic of only theoretical interest. Recent fundamental advances in graph theory, however, have made available powerful new *nonconstructive* tools that can be applied to guarantee membership in  $P$ . These tools are nonconstructive at two distinct levels: they neither produce the decision algorithm, establishing only the finiteness of an obstruction set, nor do they reveal whether such a decision algorithm can be of any aid in the construction of a solution. We briefly review and illustrate the use of these tools, and discuss the seemingly formidable task of finding the promised polynomial-time decision algorithms when these new tools apply. Our main focus is on the design of efficient self-reduction strategies, with combinatorial problems drawn from a variety of areas.

**KEY WORDS:** Polynomial-time complexity, search problems, self-reducibility, well-partial-order theory.

**C.R. CATEGORIES:** F.2.2. [Analysis of algorithms]; G.2.2. [Discrete mathematics]

### 1. INTRODUCTION

A central concern of concrete complexity theory has been establishing the boundaries of  $P$ , that is, determining exactly those problems that are decidable in

\*This research has been supported in part by the National Science Foundation under grants ECS-8403859 and MIP-8603879, by the Joint Services Electronics Program under contract N00014-84-C-0149, and by the Office of Naval Research under contracts N00014-88-K-0343 and N00014-88-K-0546.

(deterministic) polynomial time. Decision problems have traditionally been shown to be in  $P$  by producing efficient algorithms that actually attempt to solve "search problems" [13], constructing solutions to optimization versions of the problems whenever such solutions exist. Thus, while the distinction between decision and construction has remained well-defined in the study of NP-completeness, this same distinction has until now been rather blurred for problems in  $P$ .

This situation is suddenly and dramatically altered, however, as a consequence of powerful, easy-to-apply graph theory tools recently made available by the seminal work of Robertson and Seymour. See, for example, [27, 28, 29, 30]. When these tools can be employed, they classify problems as decidable in polynomial time by proving merely the *existence* of polynomial-time *decision* algorithms [6, 7, 8]. More importantly, for the subject of this paper, there is no guarantee that an optimal solution can in fact be *constructed* in polynomial time, even if an efficient decision algorithm is found. These developments, therefore, call in to question the previous folk wisdom that only the complexity of decision problems need be addressed [21]. Moreover, they motivate a serious study of efficient strategies for employing decision algorithms to construct solutions to concrete problems. This process, termed *self-reducibility*, has until now been primarily a subject of theoretical interest in investigating complexity classes [19, 24, 31].

In the next section, we briefly review the background material necessary to state the relevant results from graph theory. In Section 3, we demonstrate their use by means of a simple example. In Section 4, we establish the self-reducibility of four illustrative problems, in two cases improving on the best previously-known bounds, in two cases proving polynomial-time constructivity for the first time. Section 5 consists of a collection of general remarks pertinent to future research on this topic. We also mention some other problems amenable to this nonconstructive approach that we have not been able to self-reduce and discuss possible explanations for this difficulty.

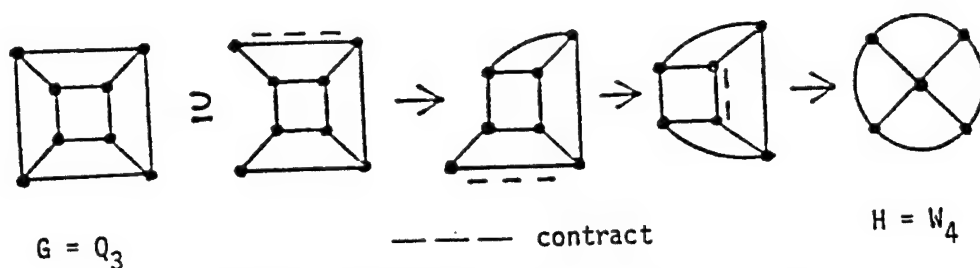
## 2. THEORETICAL FOUNDATIONS

In this section, we first give some basic definitions in order to enable us to state a fundamental theorem due to Robertson and Seymour. All graphs we consider are finite and undirected, and may have loops or multiple edges.

Given a graph  $G=(V, E)$ , a graph  $H$  is a *minor* of  $G$ , denoted  $H \leq G$ , if  $H$  can be obtained from a subgraph of  $G$  by contracting edges. For example, the graph of the wheel with four spokes is a minor of the graph of the three-dimensional binary cube, as can be seen by the construction depicted in Figure 1 (other constructions suffice as well for this example).

Note that the relation  $\leq$  defines a partial ordering on finite graphs. A family  $F$  of finite graphs is said to be *closed* under minors if, whenever  $G$  is in  $F$ , every minor of  $G$  must also be in  $F$ . Robertson and Seymour [29] have shown that, for every *fixed* graph  $H$ , the problem that takes as input a graph  $G$  and determines whether  $H \leq G$  is solvable in polynomial time.

Kuratowski's Theorem [20] can be restated as follows: a graph  $G$  is planar if

Figure 1 Construction demonstrating that  $W_4$  is a minor of  $Q_3$ .

and only if neither  $K_5 \leq G$  nor  $K_{3,3} \leq G$ . The *obstruction set* for a minor-closed family  $F$  of finite graphs is the set of graphs in the complement of  $F$  that are minimal in the minor ordering. Answering in the affirmative a long-unresolved conjecture due to Wagner [32], Robertson and Seymour have shown [30] that any set of finite graphs contains only a *finite* number of minor-minimal elements. (In other words, graphs are *well-partially-ordered* by minors.) This result is *inherently nonconstructive* in the following sense [10]: there can be no systematic method for computing the finite obstruction set for an arbitrary minor-closed family  $F$  from the description of a Turing machine that accepts precisely the graphs in  $F$ . Therefore, although we are assured of a finite obstruction set, the proof of the theorem gives no information about how to find the set, how to determine its cardinality, or even how to bound the order of the largest graph it contains.

For our work in this paper, we summarize the above results and state the following, denoting it as the RS (Robertson–Seymour) Theorem.

**RS THEOREM** Any family of finite graphs that is closed under minors can be recognized in polynomial time—specifically, in  $O(|V|^3)$  time. Moreover, if the family excludes a planar graph, then membership can be decided in  $O(|V|^2)$  time.

### 3. A SIMPLE APPLICATION OF THE RS THEOREM

To illustrate the use of the RS Theorem, we consider the following fixed-parameter variant of the NP-complete *longest path* problem [13]. In this *k longest path* (kLP) problem, we are given a graph  $G$  and are asked whether  $G$  contains a simple path with  $k$  or more edges. Obviously, this variant can be solved by brute force in polynomial time. We simply check all  $\binom{|V|}{k+1}(k+1)!$  possible paths of length  $k$ . Less directly, we now show that, as an immediate corollary to the RS Theorem, there must exist a low-order polynomial-time algorithm.

**THEOREM 2.1** The kLP problem can be decided in  $O(|V|^2)$  time.

*Proof* Observe that the family of “yes” instances for kLP is *not* closed under minors, because taking a subgraph or contracting edges reduces the size of  $G$  and so can eliminate long paths. Fortunately, however, the family of “no” instances is closed under minors. That is, if  $G$  has no simple path of length greater than or

equal to  $k$ , then taking a subgraph or contracting edges could never create one. Thus the RS Theorem applies. Moreover, the obstruction set for this simple problem has only one element, namely, the path of length  $k$ . This is certainly planar, so  $k$ LP can be decided in  $O(|V|^2)$  time.  $\square$

Of course, all we have argued is the existence of a quadratic-time algorithm for the *decision* version of  $k$ LP. We seek to know whether Theorem 2.1 can be used to tell us anything about how long it takes to *construct* a path of length  $k$  or more if any exist. As we shall show in the next section, it turns out that  $k$ LP is self-reducible.

#### 4. POLYNOMIAL-TIME SELF-REDUCIBILITY

**THEOREM 3.1** *A solution to the  $k$ LP problem can be constructed, if any exist, in  $O(|V|^2 \log |V|)$  time.*

*Proof* For the purpose of illustration, we first describe a simple method to construct a solution in  $O(|V|^4)$  time. In general, we treat the decision algorithm much like an oracle. Suppose we are given a graph  $G$  for which the decision algorithm tells us that there is a path of length at least  $k$ . We delete some edge,  $e$ , (and any multiple copies of  $e$ ) and again ask the decision question, this time on  $G - \{e\}$ . If the answer is "yes", then (all copies of)  $e$  can be discarded and the remaining graph still has a path of length at least  $k$ . On the other hand, if the answer is "no", then that edge is needed to build a path of length  $k$ . Therefore, by calling the decision algorithm at most  $|E| - 1$  times, we have located an appropriate path. (Notice that the order in which edges are considered may affect which path is discovered when  $G$  contains multiple solutions.)

In order to self-reduce more efficiently, we arbitrarily partition  $V$  into  $k+2$  classes  $V_i$ ,  $1 \leq i \leq k+2$ , of cardinalities as equal as possible. (That is,  $\|V_i\| - \|V_j\| \leq 1$  for  $1 \leq i, j \leq k+2$ .) Since  $G$  contains a path of length  $k$  on  $k+1$  vertices, so does  $G - V_s$  for some  $s$ ,  $1 \leq s \leq k+2$ . Such an  $s$  can be determined by deciding at most  $k+2$  instances of  $k$ LP. By repeating this procedure, next on  $G - V_s$ , we can reduce  $G$  to a subgraph  $G'$  of order  $k+1$  containing a path of length  $k$  in  $O(\log |V|)$  steps. At this point, there is only a constant amount of work left to do to identify an appropriate path. Since each call to the decision algorithm requires  $O(|V|^2)$  time, the time bound in the statement of the theorem is assured.  $\square$

For sufficiently dense graphs, this improves on the lowest previously-published upper bound (both for decision and construction) of  $O(|V||E|)$  for  $k$ LP [23].

Although things have gone nicely for this simple example, we remind the reader that the RS Theorem is nonconstructive. No general method for isolating obstruction sets is known. Moreover, even if an obstruction set were available, there is absolutely no guarantee that computing an optimal solution can be accomplished within any given time-complexity class. Therefore, we must be aware of the (counter-intuitive) possibility that there may exist natural problems whose decision versions never need more than low-order polynomial time while their

seemingly closely-related optimization versions require, in the worst-case at least, exponential or worse time!

Consider now the *minimum cut linear arrangement* problem [13], which is NP-complete even when input is restricted to planar graphs of maximum degree three [25], but can be solved in  $O(|V|\log|V|)$  time for trees [33]. In the relevant fixed-parameter variant of this problem (*kMCLA*), we are given a graph  $G$  and are asked whether there is an arrangement of the vertices of  $G$  along a horizontal line so that any vertical line placed between consecutive vertices cuts at most  $k$  edges connecting vertices on opposite sides of the vertical line.

In [22], it has been demonstrated that *kMCLA* is solvable in  $O(|V|^{k-1})$  time by dynamic programming. In [8], it has been shown that the RS Theorem can be applied to guarantee that *kMCLA* is decidable in  $O(|V|^4)$  time. This result relies on a cost-preserving transformation [25] that reduces an arbitrary graph of order  $|V|$  to an equivalent graph of maximum degree three with order  $O(|V|^2)$ . It remains only to show [8] that the family of graphs of maximum degree three that are "yes" instances of *kMCLA* is closed under minors. Hence we face the issue of self-reducibility.

**THEOREM 3.2** *A solution to kMCLA can be constructed, if any exist, in  $O(|V|^6)$  time.*

*Proof* Assume that  $G$  is connected and that *kMCLA*( $G$ ) = "yes". Obviously, no two vertices have more than  $k$  edges between them.

We first find a vertex that can serve as a starting point (leftmost or rightmost vertex) in a satisfactory arrangement. To do this, we choose an arbitrary vertex  $y$  of  $V$ , add a new vertex  $x$ , and augment  $E$  with  $k$  edges between  $x$  and  $y$ , thereby obtaining a new graph  $G'$ . Clearly, for some choice of  $y$ , it must be that *kMCLA*( $G'$ ) = "yes", implying that all permissible arrangements of  $G'$  begin with  $x$ , then  $y$  (the connectedness of  $G$  rules out all other possibilities). Finding such a vertex  $y$  requires at most  $O(|V|^5)$  time.

We now build on  $G'$  to find a second vertex. To accomplish this, we choose an arbitrary vertex  $z$  of  $V' - \{x, y\}$ , augment  $E'$  as necessary so that there are  $k$  edges between  $y$  and  $z$ , and replace each edge of the form  $(y, a)$ ,  $a \notin \{x, z\}$ , with  $(z, a)$ , thereby obtaining a new graph  $G''$ . It follows that, for some choice of  $z$ , it must be that *kMCLA*( $G''$ ) = "yes", implying that all permissible arrangements for  $G''$  begin with  $x$ , then  $y$ , then  $z$ . Conversely, a permissible arrangement of the vertices of  $G''$  is a permissible arrangement for  $G'$  as well. Finding such a vertex  $z$  requires  $O(|V|^5)$  time.

We repeat this construction, each time modifying the graph so as to "freeze" a prefix of some satisfactory arrangement, producing a solution in  $O(|V|^6)$  time.  $\square$

We now move on to an important NP-complete *non-graph* problem [26] that has been the subject of much study in the VLSI community. This fundamental, combinatorial problem lies at the heart of a number of circuit layout styles, and has accordingly acquired several names, most notably *gate matrix layout* and *multiple PLA folding*. (We adopt the former.) An instance of the *gate matrix layout* problem consists of a set of  $n$  nets (rows) and their respective connections to a set of  $m$  gates (columns). The goal is to find a permutation of the gates that permits

the circuit to be laid out in  $k$  or fewer tracks. More formally, our fixed-parameter variant can be described as follows.

**k gate matrix layout (kGML)**

Instance: An  $n \times m$  Boolean matrix  $M$ .

Question: Can the columns of  $M$  be permuted in such a way that, if in each row we change to 1 every 0 lying between the row's leftmost and rightmost 1's, then no column contains more than  $k$  1's?

The  $k$ GML problem appears exceedingly difficult. For any  $k \geq 2$ , there are instances of  $k$ GML with only two satisfactory permutations and  $m! - 2$  unsatisfactory ones [5], precluding any polynomial-time brute-force attack that focuses on a predetermined set of column permutations. Surprisingly, however, it has been shown in [6] that the matrix  $M$  can be modeled as a graph  $G(M)$  on  $n$  vertices so that the family of graphs corresponding to "yes" instances is closed under minors for every fixed  $k$ . Moreover, planar obstructions exist for each  $k$  [6]. Therefore, we know that the RS Theorem is applicable and  $k$ GML is decidable in  $O(n^2)$  time. (Incidentally, for  $k=2$ , it is known that there are only two simple obstructions. For  $k=3$ , the size of the obstruction set has been bounded and its elements, now numbering at least 110, are being enumerated [2]. Determining obstruction sets for  $k \geq 4$  is, as of this writing, an open problem.)

Recall, however, that our goal is to find a satisfactory permutation of the columns of  $M$ , if any exist. (Given such a permutation, it is easy to complete the track assignments of the layout with a simple greedy rule [16, 26].) Inspired by the existence of the aforementioned decision algorithm, we now show that  $k$ GML is self-reducible.

**THEOREM 3.3** *A solution to  $k$ GML can be constructed, if any exist, in  $O(n^3m)$  time.*

*Proof* Suppose  $k$ GML( $M$ ) = "yes". We then attempt to modify  $M$  by executing a subprogram described in pidgin Algol as follows:

```

for i ← 1 to n do
  for j ← 1 to m do
    if  $M(i, j) = 0$  then begin
       $M(i, j) \leftarrow 1$ 
      if  $k$ GML( $M$ ) = "no" then  $M(i, j) \leftarrow 0$ 
    end begin
  end do
end do.
```

Therefore, after processing all of  $M$ ,  $k$ GML( $M$ ) = "yes" still holds; no remaining 0 can be changed to a 1. The resultant matrix must now enjoy the "consecutive 1s

property". That is,  $M$  now possesses a column permutation such that, for every row, the leftmost and rightmost 1s are not separated by even a single 0. We employ the PQ-Tree algorithm as described in [3, 14] to identify such a permutation in  $O(nm)$  time. (Notice that the solution so obtained may be dependent on the order in which the elements of  $M$  were considered.)  $\square$

We conclude this section with a generic form of fixed-parameter problem. Let  $F$  denote an arbitrary minor-closed family of graphs.

within  $k$  vertices of  $F$  ( $wkvF$ )

Instance: A graph  $G$ .

Question: Does  $G$  contain a set of  $k$  or fewer vertices that, when deleted, leave a graph in  $F$ ?

As examples,  $k$ -vertex cover and  $k$ -feedback vertex set are representatives of  $wkvF$  when  $F$  is the family of edgeless graphs and the family of acyclic graphs, respectively. Also, when  $F$  is the family of planar graphs, the notion of "planarizing sets" has been addressed in [12, 15].

For every fixed valued of  $k$  and every minor-closed family,  $F$ , this problem is trivially decidable in polynomial time by brute force. One need only check the  $\binom{|V|}{k}$  graphs that result from the removal of  $k$  vertices. This can be done in  $O(|V|^k p(|V|))$  time, where  $p(|V|)$  bounds the time required to test for membership in  $F$ . However, it has been shown in [7] that  $wkvF$  can in fact be decided in only  $O(|V|^3)$  time. We shall now prove that this permits a low-order polynomial-time self-reduction strategy as well, enabling the construction of a solution in less than the  $O(|V|^k p(|V|))$  time bound given by brute force.

**THEOREM 3.4** *A solution to  $wkvF$  can be constructed, if any exist, in  $O(|V|^4)$  time.*

*Proof* Let  $H$  denote some fixed member of the (finite) obstruction set for  $F$ , and suppose  $wkvF(G) = \text{"yes"}$ . We now identify an arbitrary vertex,  $x$ , of  $G$  with an arbitrary vertex,  $y$ , of  $H$ . That is, we construct  $G' = \langle V_{G'}, E_{G'} \rangle$  where

$$V_{G'} = (V_G - \{x\}) \cup (V_H - \{y\}) \cup \{z\}$$

and

$$E_{G'} = (E_G - \{\text{edges incident to } x\}) \cup (E_H - \{\text{edges incident to } y\})$$

$$\cup \{e = (a, z) \mid (a, x) \in E_G \text{ or } (a, y) \in E_H\}.$$

(Multiple copies of edges can be discarded.) Since  $H$  is a minor-minimal element of  $\bar{F}$ , the removal of exactly one vertex from our copy of  $H$  in  $G'$  is now necessary; it may as well be  $z$ . It follows that we can use  $x$  in a solution for  $G$  if and only if  $wkv(G') = \text{"yes"}$ . Since  $H$  is fixed,  $|V_{G'}|$  is  $O(|V|)$  and we can decide  $wkvF(G')$  in at most  $O(|V|^3)$  time.

After considering  $x$ , we move on to employ this construction with another copy

of  $H$  at yet another vertex of  $G$ , building on  $G'(G)$  if  $wkv(G) = \text{"yes"}$  ("no"). By the time we have considered each vertex in  $G$ , we must have isolated a  $k$ -vertex solution, satisfying the time bound in the statement of the theorem.  $\square$

## 5. CONCLUDING REMARKS

The nonconstructive methods that have motivated our investigations in this paper raise serious questions about the relationship between algorithms that decide and algorithms that construct or optimize. This relationship has in the past often been conveniently (and mistakenly) taken for granted.

An interesting aspect of these tools based on well-partially-ordered sets is their power to demonstrate polynomial-time complexity bounds for some problems for which there is no known alternate proof available for membership in either NP or co-NP, and in some cases, for which no alternate proof is known even for decidability! It is just such problems that appear at present to resist the kinds of self-reduction strategies that we have illustrated herein.

For example, consider the problem of finding, when any exist, an embedding of a graph  $G$  in 3-space so that no cycle of  $G$  is nontrivially knotted [4, 7]. The family of graphs that have such *knotless* embeddings is closed under minors, and therefore the problem of deciding whether a graph has any such embedding can be solved in  $O(|V|^3)$  time. No alternate proof of decidability is known; the difficulties of establishing such a proof appear to be significant [17, 18]. Similarly, no self-reduction algorithm is known; the problem of devising one appears to be formidable indeed.

The impact of these nonconstructive methods on the fields of complexity theory and algorithm design remains to be determined. From a theoretical standpoint, they inspire a study of constructive complexity [1] and a formal notion of fast self-reducibility [9]. Also, new techniques for identifying polynomial-time algorithms without the need for entire obstruction sets have begun to appear [10]. If useful decision algorithms are possible for some problems amenable to this general approach, then the existence and efficiency of self-reduction algorithms may become an increasingly practical issue as well.

## References

- [1] K. Abrahamson, M. R. Fellows, M. A. Langston and B. Moret, Constructive Complexity, Computer Science Technical Report CS-88-191, Washington State University, 1988.
- [2] R. L. Bryant, M. R. Fellows, N. G. Kinnersley and M. A. Langston, On finding obstruction sets and polynomial-time algorithms for gate matrix layout, *Proc. 25th Allerton Conf. on Communication, Control, and Computing* (1987), 397-398.
- [3] K. S. Booth and G. S. Lueker, Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms, *J. of Computer and System Sciences* 13 (1976), 335-379.
- [4] J. Conway and C. Gordon, Knots and links in spatial graphs, *J. Graph Th.* 7 (1983), 445-453.
- [5] N. Deo, M. S. Krishnamoorthy and M. A. Langston, Exact and approximate solutions for the gate matrix layout problem, *IEEE Trans. on Computer-Aided Design* 6 (1987), 79-84.
- [6] M. R. Fellows and M. A. Langston, Nonconstructive advances in polynomial-time complexity, *Info. Proc. Letters* 26 (1987), 157-162.

- [7] M. R. Fellows and M. A. Langston, Nonconstructive tools for proving polynomial-time decidability, *J. of the ACM* 35 (1988), 727-739.
- [8] M. R. Fellows and M. A. Langston, Layout permutation problems and well-partially-ordered sets, *Proc. 5th MIT Conf. on Advanced Research in VLSI* (1988), 315-327.
- [9] M. R. Fellows and M. A. Langston, Fast self-reduction algorithms for combinatorial problems of VLSI design, *Proc. 3rd Aegean Workshop on Computing* (1988), 278-287.
- [10] M. R. Fellows and M. A. Langston, On search, decision and the efficiency of polynomial-time algorithms, *Proc. 21st ACM Symp. on Theory of Computing* (1989), 501-512.
- [11] H. Friedman, N. Robertson and P. D. Seymour, The meta mathematics of the graph minor theorem, in: *Applications of Logic to Combinatorics*, American Math. Soc., Providence, RI, to appear.
- [12] J. R. Gilbert, J. P. Hutchinson and R. E. Tarjan, A separator theorem for graphs of bounded genus, *J. of Algorithms* 5 (1984), 391-407.
- [13] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, CA, 1979.
- [14] M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, NY, 1980.
- [15] J. P. Hutchinson and G. L. Miller, On deleting vertices to make a graph of positive genus planar, *Prospectives in Computing* 15 (1987), 81-98.
- [16] A. Hashimoto and J. Stevens, Wire routing by optimizing channel assignment within large apertures, *Proc. 8th Design Automation Workshop* (1971), 155-169.
- [17] W. Jaco, private communication.
- [18] D. S. Johnson, The many faces of polynomial time, in *The NP-Completeness Column: An Ongoing Guide*, *J. Algorithms* 8 (1987), 285-303.
- [19] R. M. Karp and R. J. Lipton, Some connections between nonuniform and uniform complexity classes, *Proc. 12th ACM Symp. on Theory of Computing* (1980), 302-309.
- [20] C. Kuratowski, Sur le probleme des courbes gauches en topologie, *Fund. Math.* 15 (1930), 271-283.
- [21] R. M. Karp, E. Upfal and A. Wigderson, Are search and decision problems computationally equivalent, *Proc. 17th ACM Symp. on Theory of Computing* (1985), 464-475.
- [22] F. S. Makedon and I. H. Sudborough, On minimizing width in linear layouts, to appear.
- [23] B. Monien, How to find long paths efficiently, *Annals of Disc. Math.* 25 (1985), 239-254.
- [24] A. Meyer and M. Paterson, With what frequency are apparently intractable problems difficult, Technical Report, MIT, 1979.
- [25] B. Monien and I. H. Sudborough, Min cut is NP-complete for edge weighted trees, *Lecture Notes in Computer Science* 226 (1986), 265-274.
- [26] T. Ohtsuki, H. Mori, E. S. Kuh, T. Kashiwabara and T. Fujisawa, One dimensional logic gate assignment and interval graphs, *IEEE Trans. on Circuits and Systems* 26 (1979), 675-684.
- [27] N. Robertson and P. D. Seymour, Disjoint paths—a survey, *SIAM J. Alg. Disc. Meth.* 6 (1985), 300-305.
- [28] N. Robertson and P. D. Seymour, Graph minors—a survey, in: *Surveys in Combinatorics* (I. Anderson, ed.), Cambridge University Press, 1985.
- [29] N. Robertson and P. D. Seymour, Graph Minors XIII. The Disjoint Paths Problem, to appear.
- [30] N. Robertson and P. D. Seymour, Graph Minors XVI. Wagner's Conjecture, to appear.
- [31] C. P. Schnorr, Optimal algorithms for self-reducible problems, *Proc. ICALP* (1976), 322-337.
- [32] K. Wagner, Über einer eigenschaft der ebener complexe, *Math. Ann.* 14 (1937), 570-590.
- [33] M. Yannakakis, A polynomial algorithm for the min cut linear arrangement of trees, *J. of the ACM* 32 (1985), 950-959.

# The University of Tennessee

---

Michael A. Langston, Department of Computer Science, Knoxville, Tennessee 37996-1301  
phone: (423) 974-3534; fax: (423) 974-4404; email: langston@cs.utk.edu

March 29, 1999

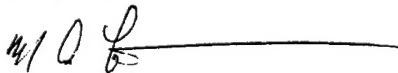
Dr. Ralph F. Wachter  
Office of Naval Research  
800 North Quincy Street, Ballston  
Tower One  
Arlington, VA 22217-5660

Dear Dr. Wachter,

I would like to thank ONR for its support of my work under grant N00014-90-J-1855.

A final technical report is enclosed.

Sincerely yours,



Michael A. Langston  
Professor

ML/~~La~~TeX

enclosures: final report in triplicate

cc: Atlanta Office (1)

NRL (1)

~~DMIC~~ (2)